Fortran grammatica-extractie

Jan Derriks

5 september 2007

Master Software Engineering

Afstudeerdocent : P. Klint

Stagebegeleider: J. Vinju

Opdrachtgever: P. Klint

Publicatiestatus: openbaar

Universiteit van Amsterdam,
Hogeschool van Amsterdam,
Vrije Universiteit

Inhoudsopgave

Samenvatting

Dit verslag beschrijft de extractie van een Fortran grammatica met als doel een bruikbare basis-grammatica in SDF formaat (Syntax Definition Formalism) beschikbaar te stellen voor de Meta-Environment – een framework voor code analyse en transformatie dat is ontwikkeld door het CWI (Centrum voor Wiskunde en Informatica).

Als bron voor de extractie is de ISO standaard documentatie gebruikt voor de versie van Fortran die Fortran90 wordt genoemd en een bestaande Fortran grammatica die onderdeel vormt van het ELI project; dit is een verzameling tools voor compiler constructie die publiek beschikbaar is.

De eerste vraag is of de opmaak van Fortran code niet te complex is om door een (via de ASF+SDF Meta-Environment gegenereerde) parser op basis van SDF gelezen te kunnen worden zonder pre-processing op de broncode. Dit blijkt lastig maar niet onmogelijk. Betere alternatieven dan de gebruikte oplossing worden genoemd die een aanpassing van de Meta-Environment vergen.

De tweede vraag is hoe de kwaliteit en buikbaarheid van de geproduceerde grammatica kan worden aangetoond. De begrippen bruikbaarheid en kwaliteit van een grammatica worden toegelicht. De bruikbaarheid van de geproduceerde Fortran grammatica voor code-analyse wordt aangetoond en er wordt beschreven hoe een beperkte coverage test op de grammatica en de gebruikte testcode wordt uitgevoerd om de grammatica beter te kunnen testen.

De resulterende SDF Fortran grammatica bestaat uit 768 productieregels en is verdeeld in 11 modulen die nu onderdeel vormen van de SDF taalbibliotheek van de Meta-Environment.

Voorwoord

Dank aan mijn begeleiders, Paul Klint en Jurgen Vinju voor de hulp en het beschikbaar stellen van een opdracht en werkplek op het befaamde Centrum van Wiskunde en Informatica.

Dank ook aan Marjan Freriks, voormalig directeur van het Instituut van Informatica van de Hogeschool van Amsterdam, die het medewerkers van het IvI mogelijk heeft gemaakt om een master opleiding te volgen.

In memoriam John Backus (3 dec 1924 - 17 maart 2007), de "vader"van zowel Fortran als BNF.

Hoofdstuk 1

Inleiding

De ASF+SDF Meta-Environment [1] van het CWI [2] bevat een verzameling tools waarmee tools voor diverse (programmeer-)omgevingen worden ontwikkeld. Het is een programmeeromgeving om programmeromgevingen te bouwen, vandaar de naam 'meta-omgeving'. Voorbeelden van toepassingen zijn programma-analyse (metrieken bepalen bijvoorbeeld), softwarerenovatie of compiler constructie. De Meta-Environment wordt geleverd met een bibliotheek van grammatica-beschrijvingen van enkele programmeertalen zoals C, Java en Pico. Deze grammatica's zijn beschreven in het SDF formaat (Syntax Definition Formalism), dat lijkt op EBNF (Extended Backus Naur Form).

Een grammatica voor de taal Fortran had het CWI nog niet. De ontwikkeling hiervan is het onderwerp van dit verslag. Fortran is een taal zonder gereserveerde keywords en met specifieke opmaakregels die het een scanner/parser bijzonder lastig kunnen maken. Spaties zijn bijvoorbeeld niet noodzakelijk in Fortran77. In hoofdstuk 3.3 meer over de taal Fortran.

In hoofdstuk 2 wordt de probleemstelling beschreven. Een van de vragen die bij dit project naar voren komt is in hoeverre het ontwikkelen van een grammatica te vergelijken is met reguliere software ontwikkeling. Denk hier bij ontwikkelen niet aan nieuwbouw; het gaat hier om het reconstrueren van een bestaande grammatica die echter niet in de gewenste vorm beschikbaar is. Ook de standaard ISO Fortran beschrijving is niet direct te gebruiken,

[1] http://www.meta-environment.org

[2] Centrum voor Wiskunde en Informatica – http://www.cwi.nl

al is dat wel een goed uitgangspunt.

Hoofdstuk 3 geeft achtergrondinformatie over wat *grammar recovery* en *grammarware engineering* wordt genoemd. Ook de diverse Fortran standaarden en versies worden in dit hoofdstuk toegelicht.

In hoofdstuk 4 wordt de onderzoeksmethode beschreven. Hoofdstuk 5 is een uitwerking van het onderzoek. Paragraaf 5.2 beschrijft het testen van de grammatica, zowel met- als zonder gebruik van bestaande Fortran code als testdata.

Als slot volgen het resultaat en de conclusie. In de bijlage is de complete grammatica in klein lettertype opgenomen. De laatste versie hiervan is te vinden in het versiebeheersysteem van de Meta-Environment.

In dit document wordt Engels en Nederlands door elkaar gebruikt om aan te sluiten bij het vakjargon wat bij taaltechnologie gebruikt wordt. Termen als *ge-parsed* worden niet vertaald naar *geparseerd*. In dat opzicht voldoet dit document niet aan de Nederlandse grammatica. Hopelijk kan de lezer het toch scannen en parsen.

Hoofdstuk 2

Probleemstelling

De doelstellingen van dit project zijn drieledig: enerzijds is er de behoefte om een bruikbare Fortan grammatica toevoegen aan de SDF-bibliotheek in de Meta-Environment. Anderzijds is de ontwikkeling/extractie van deze grammatica een geschikt project om diverse aspecten over de kwaliteit van een grammatica te gaan onderzoeken: waaraan kan de kwaliteit van een (SDF) grammatica worden afgemeten en hoe kan dit worden getest? Deze vraag zal worden beantwoord met de geproduceerde Fortran grammatica als case. Als derde doel is het ontwikkeltraject zelf onderwerp van onderzoek: is grammarware ontwikkeling vergelijkbaar met "normale" software ontwikkeling?

De eisen (requirements) waaraan de Fortran SDF grammatica dient te voldoen zijn tevoren niet scherp vastgelegd. Een tamelijk willekeurig gekozen kwaliteitsmaat is dat het mogelijk moet zijn om minstens 200.000 regels Fortran code te kunnen parsen om daar bijvoorbeeld een call graph van te kunnen bepalen. Dit is slechts een van de mogelijke use-cases van een grammatica.

Er zijn diverse subvragen die bij dit project gesteld kunnen worden:

1. Is het überhaupt mogelijk om een SDF Fortran definitie te maken waaruit een parser gegenereerd kan worden? Vooral de oudere Fortran versies kennen zeer specifieke opmaakregels die het een parser erg lastig kunnen maken.

2. Fortran is vastgelegd in ISO standaarden. Is de ISO Fortran standaard

een goed uitgangspunt voor een grammatica? Zoja, in hoeverre voldoet de resulterende SDF grammatica aan de ISO standaard?

3. Wat bepaalt de bruikbaarheid en kwaliteit van een grammatica?

4. Hoe test je de kwaliteit en bruikbaarheid van de geproduceerde grammatica?

Op deze vragen zal een antwoord worden gegeven. Het ontwikkelen van een bruikbare Fortran SDF grammatica voor de Meta-Environment is het hoofddoel van dit project. De belangrijkste onderzoeksvraag is vervolgens hoe de kwaliteit van deze grammatica kan worden bepaald.

Hoofdstuk 3

Achtergrond

3.1 Grammarware en grammarware hacking

Een grammatica beschrijft de structuur van een taal. Een taal kan bestaan zonder grammaticabeschrijving, net zoals een programma kan bestaan zonder ontwerpbeschrijving. Er zijn heel veel talen in de wereld en er komen dagelijks talen bij. Denk hierbij aan MSN-taal of SMS-taal, programmeertalen of datacommunicatie-talen. Het is interessant om de ontwikkeling van menselijke talen en computertalen met elkaar te vergelijken maar we beperken ons hier verder tot de computertalen - dit is immers de basis waarop het vakgebied software engineering drijft. Door computers leesbare (programmeer-)talen zijn *context-vrije* talen en kunnen worden beschreven in een Context Free Grammar (CFG) in de vorm van een verzameling BNF of SDF productieregels.

De grammatica van een computertaal is meestal te achterhalen aan de hand van een compiler, handboek of een meer of minder formeel vastgelegde taalbeschrijving zoals bij Fortran. Een compiler of interpreter is misschien het meest bekende maar zeker niet niet het enige tool waarvoor een grammatica noodzakelijk is. Enkele andere voorbeelden van tools die een grammatica nodig hebben:

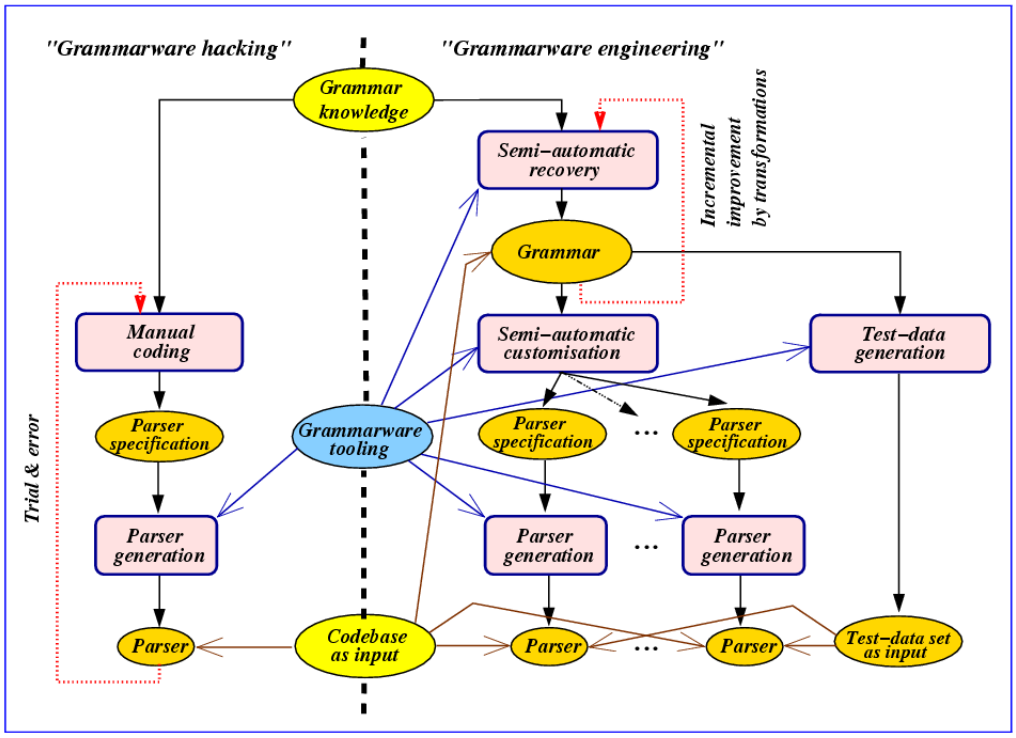
- internet browsers (kennis van Html, XML)
- datacommunicatie-tools (anders dan een browser, kennis van een specifiek dataformaat en protocol)

- syntaxgevoelige editors (pretty-printing en highlighting)
- data-transformatietools
- data-analysetools
- code-analyse tools (debuggers, metriek-tools e.d.)
- code-renovatie tools

Een mooi voorbeeld van code-renovatie waarbij grammaticakennis wordt benut is de software refactoring feature die vaak in ontwikkelomgevingen is ingebouwd en waarmee bijvoorbeeld in één bewerking de naam van een functie kan worden gewijzigd inclusief alle aanroepen van deze functie maar niet binnen commentaar of stringconstanten. Een simpele zoek-en-vervang opdracht kan dit niet.

Voor dit voorbeeld is niet *alle* grammaticakennis van een taal nodig – een subset van de grammatica zou volstaan. Verschillende toepassingen stellen dus verschillende eisen aan de grammatica. In [4] wordt het doel van een grammatica een *grammar use case* genoemd, en met de term *base-line grammar* wordt de grammatica bedoeld waarvan specifieke grammar use cases kunnen worden afgeleid. Om zo'n de base-line grammar geschikt te maken voor de betreffende use case wordt deze aangepast, uitgebreid of juist vereenvoudigd tot de doel-grammatica (*enriched grammar*).

Software dat gebruik maakt van een grammatica (zoals een compiler) kunnen we *grammarware* noemen. Software om deze software te maken wordt *meta-grammarware* genoemd [4]. De ASF+SDF Meta-Environment is een voorbeeld van meta-grammarware. Hiermee kan een parser voor een taal gegenereerd kan worden waarmee bijvoorbeeld een pretty-printer kan worden gemaakt. Belangrijker is echter de mogelijkheid om onderhoud te kunnen doen op bestaande broncode door middel van automatische transformaties. Steeds vaker zal het onmogelijk zijn om grote software systemen handmatig te renoveren als dit nodig is. Redenen voor transormatie zijn bijvoorbeeld de euro-conversie, millenium-conversie (Y2K), taalconversie, bedrijfsfusie, enz. Om automatische conversie van code voor diverse talen mogelijk te maken is voor elk van deze talen een grammatica noodzakelijk. Grammatica's voor renovatie-doeleinden worden ook wel *renovation-grammars* genoemd [6]. De ASF+SDF Meta-Environment kan teksttransformaties uitvoeren door gebruik te maken van herschrijf-instructies die geschreven worden in ASF (*Algebraic Specification Formalism*) regels.



Figuur 3.1: Parser development as an example of grammarware hacking vs engineering (uit: [4])

3.1.1 Grammatica ontwikkelmethode

Hoe wordt grammarware ontwikkeld? Voor “normale” software ontwikkeling (non-grammarware) zijn diverse ontwikkelmethodieken beschreven, de laatste jaren vooral gericht op objectgeorienteerde methoden (zoals RUP) en een iteratieve of ’agile’ aanpak (extreme programming). Er zijn ook talloze ontwikkeltools beschikbaar die de programmeur ondersteunen bij zowel ontwerp, bouw als testen van applicaties. In het artikel *Toward an engineering discipline for Grammarware* [4] wordt gesteld dat testen, design, implementatie en recovery van grammarware nog vooral ad-hoc gebeurt, op een manier die je *grammarware hacking* kunt noemen (zie fig 3.1). Aspecten uit bestaande ontwikkelmethoden zoals modularisatie en (unit-)tests kunnen uiteraard ook worden toegepast bij grammarware ontwikkeling. De ASF+SDF Meta-Environment biedt hier enige ondersteuning voor. Er kan nog veel meer gedaan worden om grammarware-ontwikkeling te ondersteunen met tools die automatische grammatica-transformaties kunnen uitvoeren om bijvoorbeeld refactoring of style conversion (zoals van BNF naar EBNF notatie) te automatiseren.

3.2 Grammatica kwaliteit

Hoe test je een grammatica? Dat hangt af van de use-case (het doel, de requirements) van de grammatica. Moet er een compiler mee worden gemaakt of is het bedoeld om syntax-highlighting in een editor mogelijk te maken? Een testcriterium als *alle beschikbare Fortran code kunnen parsen* is met een parser gebaseerd op een minimale grammatica al te bereiken. De SDF code

```
~[]*    -> Program        %% alles wat niet niks is
[a-z]+  -> Identifier
```

zal alle willekeurige tekens als input accepteren als een *Program* als dat het startsymbool is. Tekst kunnen parsen is geen doel op zich (geen use-case), net zoals 'voldoen aan de Java syntax' nooit het doel van een Java programma zal zijn (maar wel een voorwaarde). Toch kan er nog veel over een grammatica-definitie (SDF-code) worden gezegd zonder de use-case te kennen. Net als bij gewone programmacode kan worden gesproken over layout, 'onbereikbare' (dode) code, overbodige code, enz. In bovenstaand voorbeeld is de tweede regel geheel overbodig voor het parsen van 'Program' en dat is aan de code ook wel te zien – de soort 'Identifier' is onbereikbaar vanaf het symbool 'Program'; vergelijkbaar met een functiedefinitie die nooit aangeroepen wordt. Een goede grammatica dient te voldoen aan dezelfde regels als die voor elk goed geschreven programma gelden als het gaat om heldere duidelijkheid en een structuur die overeenkomt met de betekenis van het programma of de semantiek van de grammatica.

3.2.1 Grammatica coverage testing

In *Grammar Testing* [5] beschrijft Ralf Lämmel het principe van *rule-coverage* en test set generatie uit een grammatica. Rule coverage bij een grammatica is vergelijkbaar met line coverage bepaling bij programmacode; tijdens een test-run wordt bekeken of elke regel code minstens een keer wordt aangeproken. Dit zegt iets over de kwaliteit van de test maar ook over de code. Een productieregel die nooit aangesproken wordt zou een fout in de grammatica kunnen betekenen of een overbodige regel kunnen zijn. Het kan ook betekenen dat de testdata niet genoeg variatie bevat en uitgebreid moet worden met stukken code waarbij de betreffende productie wel wordt gebruikt. Hoe goed is een test waabij 400000 regels Fortran code succesvol

kan worden geparsed maar waarbij slechts een klein deel van de grammaticaregels wordt benut? Is het niet mogelijk om een Fortran testprogramma te maken waarmee de grammatica volledig kan worden getest op succesvol parsen?

Een grammatica kan worden gebruikt als grondstof voor een generator. Bij transformaties wordt de generatie gestuurd door de te transformeren data en de transformatieregels. Dit principe zien we ook bij XSLT en XML om data in een presenteerbare vorm te gieten, of bij het gebruik van ASF regels om programmacode te herschrijven. Als we nu voor de generatie als doel stellen dat er test-code wordt geproduceerd die voldoet aan de eigen syntax en die zoveel mogelijk rule-coverage oplevert zou dit algoritme de generator kunnen sturen zonder input-data. Een codegenerator dus die random (correct parseerbare) code genereert, met als doel deze code als test-data te kunnen gebruiken voor (dezelfde) parser en om de grammatica op te beoordelen.

Er zijn algoritmen om het kortste pad van productie-afleidingen naar een nonterminal te zoeken. Voor generatie van goede test-data zou een ander algoritme moeten worden gebruikt dat juist alle mogelijke paden door de grammatica doorloopt en nonterminals verzint, vergelijkbaar met een Sinterklaas-rijm generator die alle bekende rijmvormen produceert maar met een beperkt aantal rijmwoorden. Voor een Fortran grammatica zou dit kunnen betekenen dat een generator een stuk geldige Fortran code produceert wat voldoet aan de Fortran syntax en waar alle Fortran syntax-elementen in gebruikt worden. Om het niet oneindig lang te laten worden moeten er randvoorwaarden worden gesteld aan de generatie van namen van identifiers of grootte van getallen. Het nut van deze random codegeneratie is dat de uitvoer gebruikt kan worden als unit test na refactoring van de grammaticacode: de testcode zou nog steeds moeten parsen. Anderzijds zou een extern 'orakel' een oordeel kunnen geven over de geldigheid van de code. Dit orakel kan een deskundige zijn of een andere parser die men vertrouwt zoals die van een goede Fortran compiler. Om gegenereerde code voor een compiler acceptabel te maken is veel meer werk nodig. Zaken als type-checking en het aantal argumenten bij functies in overeenstemming houden met de functiedeclaraties zijn afspraken die meestal niet in de productieregels van de grammatica kunnen worden vastgelegd.

Uit de Fortran grammatica kunnen dus Fortran programma's worden afgeleid die op correctheid kunnen worden gecontroleerd. Door handmatig in de Fortran grammatica de korste afleiding van het startsymbool *Executable-Program* te zoeken (via zo min mogelijk nonterminals en productieregels) is

te zien dat de kleinst mogelijk test voor een Fortran compiler bestaat uit een compleet programma van slechts een woord: END (zie onderstaande voorbeeld grammatica). Dit is een zeer nuttig testprogramma om de minimale werking van een compiler of parser te testen. Deze minimale test-term zou automatisch uit een grammatica kunnen rollen door een algoritme te gebruiken dat de *kortste* afleiding van de nonterminal *ExecutableProgram* zoekt. De hiervoor beoogde 'full-coverage-test-generator' zou nu juist *alle* afleidingen moeten genereren, met beperkingen om niet oneindig door te gaan. Hieronder volgt een stukje uit de Fortran 90 standaard (geen SDF syntax), waaruit is op te maken dat ook *END PROGRAM*, *END PROGRAM FOO* en *PROGRAM FOO newline END PROGRAM FOO* geldige programma's zijn. De generator zou dus een verzameling van 'geldige' verschillende programma's kunnen produceren.

```
R201 executable-program  is  program-unit [ program-unit ] ...

An executable-program must contain exactly one main-program program-unit
R202 programUnit         is  main-program
                         or  external-subprogram
                         or  module
                         or  block-data

R1101 main-program       is [ program-stmt ]
                              [ specification-part ]
                              [ execution-part ]
                              [ internal-subprogram-part ]
                            end-program-stmt
R1102 program-stmt       is PROGRAM program-name
R1103 end-program-stmt   is END [ PROGRAM ] [ program-name ] ]
```

Tussen de grammaticaregels van de Fortran standaard staan in natuurlijke taal extra constraints en aanwijzingen om de taal te beschrijven. In bovenstaand voorbeeld staat dat een executable program nooit meer dan één 'main-program' mag bevatten, terwijl de productieregel voor *executable-program* een willekeurige verzameling van program-units toelaat, waar main-program ook toe behoort.
Zie bijlage A voor een bladzijde uit de ISO standaard met elkele constraints. Dit soort opmerkingen zullen door een 'random-code-generator' uiteraard niet worden meegenomen, waardoor de kans klein is dat de generator correcte Fortran code produceert. Omgekeert geldt echter ook dat deze 'incor-

recte' code wel degelijk door de parser geaccepteerd zal worden zolang de constraints niet in de grammatica verwerkt zijn.

Dit denkbeeldige experiment (de generator is niet gebouwd) geeft inzage in de kwaliteit en de beperkingen van de grammatica en het testen van een parser die ermee gegenereerd wordt. Het maakt duidelijk dat de Fortran SDF grammatica slechts een basis is waarmee verder gewerkt kan worden. De taal Fortran is daar veel te ingewikkeld om volledig in SDF te kunnen worden vastgelegd, net als de meeste moderne talen. Daarom is de ISO definitie 'verrijkt' met talloze opmerkingen om toch te kunnen dienen als beschrijving voor compilerbouwers. De grammatica voor de Meta-Environment moet als base-line grammar worden beschouwd die verrijkt moet worden met meer informatie voor het doel van gebruiker.

3.3 De taal Fortran

Fortran is een vijftig jaar oude programmeertaal en wordt anno 2007 nog onderwezen en toegepast. Er wordt ook nog steeds gewerkt aan aanpassingen in de taal. Fortran is de eerste programmertaal die in een officiële standaard is vastgelegd – destijds door de organisatie die nu ANSI heet (American National Standards Institute). Deze Fortran standaard werd in 1966 opgesteld en is nu bekend als FORTRAN 66. Later zouden FORTRAN 77, Fortran 90, Fortran 95 en Fortran 2003 volgen. Er wordt gesproken over Fortran 2008. De eerste stap in de ontwikkeling van Fortran werd gezet in 1953 toen John Backus, werkzaam voor IBM, een voorstel indiende voor de ontwikkeling van een taal die het programmeren in het omslachtige assembly overbodig zou maken. Voor 1966 heetten deze versies resp. FORTRAN, FORTRAN II, FORTRAN III en FORTRAN IV. Deze laatste versie vormde de basis voor FORTRAN 66.

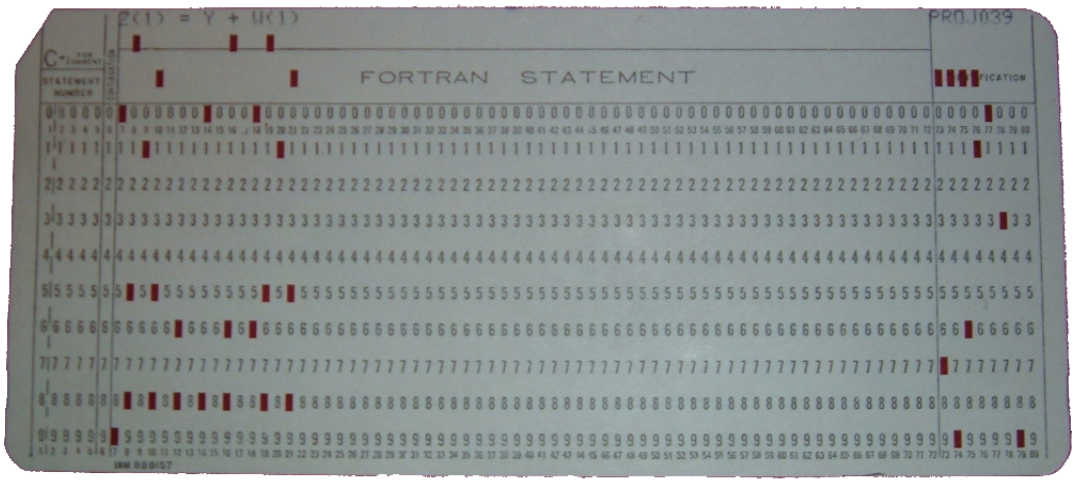
In het ISO document van de Fortran 90 standaard staat dat vanaf versie Fortran 90 het woord Fortran in de naam van de taal nu met kleine letters en een hoofdletter wordt geschreven. Dit geeft al een indicatie van de gedetailleerdheid van de standaard, die 340 bladzijden lang is. In deze scriptie zal Fortran verder met een enkele hoofdletter worden geschreven.

Terugkijkend zou je Fortran een domeinspecifieke taal kunnen noemen met als domein de exacte wetenschappen en het rekenwerk wat daarbij wordt verricht, of zoals Metcalf [7] het zegt: *Fortran's superiority had always been in the area of numerical, scientific, engineering and technical applications*

. De naam Fortran is afgeleid van de woorden Formula Translation. De taal COBOL, wat staat voor *Common Business Oriented Language*, was vooral bedoeld voor bedrijfs(administratie-) applicaties, een heel ander toepasingsgebied maar ook domeinspecifiek van oorsprong. Zo kent Cobol geen complexe getallen als standaard datatype zoals Fortran heeft. Ook in de latere ontwikkeling is Fortran de rekenpatser onder de programmeertalen gebleven. Vanaf 1992 is gewerkt aan een uitbreiding op Fortran 90 met de naam High Performance Fortran (HPF), met als doel de taal geschikt te maken voor parallelle systeemarchitecturen waarbij rekentaken door de programmeur over meerdere processoren kunnen worden verdeeld; in onderstaand voorbeeld is dit te zien aan de directives die in commentaar op de tweede en derde regel zijn verwerkt. Compilers die deze instrucies niet kennen zullen deze regels als commentaar behandelen en zo toch de code kunnen uitvoeren. De FORALL instructie is hier een Fortran 95 uitbreiding die Fortran 90 nog niet had.

```
REAL A(1000,1000)
!HPF$ PROCESSORS procs(4,4)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: A
DO k = 1, num_iter
 FORALL (i=2:999, j=2:999)
  A(i,j) = (A(i,j-1) + A(i-1,j) + A(i,j+1) + A(i+1,j))/4
 END FORALL
END DO
```

In de evolutie van de taal Fortran is ook de evolutie van de computer te herkennen. Tot Fortran 77 was de regellengte beperkt tot wat met ponskaarten kon worden ingevoerd (72 posities). Om op een tweede kaart verder te gaan dient een niet-blank teken op de 6e kolom van de kaart te worden geplaatst. De ponskaarten zijn nu regels in een tekstbestand geworden en in Fortan 77 zijn de eerste 6 posities op een regel gereserveerd voor resp. een commentaarteken, een label en een continuatie-symbool, net als op de ponskaarten. Een niet-blank teken op positie 6 in een regel betekent dat de regel een voortzetting is van de vorige. Een asterisk of C op de eerste kolom betekent dat de regel alleen commentaar bevat wat genegeerd kan worden. Deze strakke formaatbeschrijving wordt het *fixed source form* genoemd. Vanaf Fortran 90 kan ook gekozen worden voor het vrijere *free form*, waarbij commentaar ook aan het einde van een regel kan worden geplaatst, voorafgegaan door een uitroepteken. In Fortran 90 en later worden nog steeds continuatie-tekens tekens gebruikt om aan te geven dat een volgende



Figuur 3.2: Ponskaart met 72 kolommen

regel eigenlijk bij de voorgaande hoort. Onderstaande Fortran code

```
a_string = 'it is a nice day &
           &today'
```

is een voorbeeld waarbij met twee continuatie-tekens aan het einde en aan het begin van een regel tekst wordt aangegeven dat de twee regels samen het assignment statement: `a_string = 'it is a nice day today'` vormen, waarbij de beide `&` tekens en alles wat daartussen staat genegeerd wordt. Deze feature komt van pas om bijvoorbeeld de initialisatie van een array met veel waarden op een nette manier vorm te geven en wordt nog veel gebruikt.

De vooruitgang op het gebied van software constructie is ook in de evolutie van Fortran te zien. Edsger Dijkstra's beroemde artikel "Go To Statement Considered Harmful" is uit 1968, twee jaar nadat Fortran 66 uitkwam. Latere versies van Fortran maken gestructureerd programmeren mogelijk, zonder gebruik van labels en goto's. In Fortran 2003 is het mogelijk om objectgeorienteerd te programmeren door features als overerving en polymorfie. Of deze uitbreidingen het gebruik van de taal Fortran weer zal doen toenemen is onwaarschijnlijk, aangezien hele generaties inmiddels al met nieuwere talen zijn opgegroeid en Fortran alleen nog als 'een oude taal' zullen zien. Reeds in 1972 schreef Edsger Dijkstra over Fortran dat de taal verouderd

is:[1] *The second major development on the software scene that I would like to mention is the birth of FORTRAN. At that time this was a project of great temerity and the people responsible for it deserve our great admiration. It would be absolutely unfair to blame them for shortcomings that only became apparent after a decade or so of extensive usage: groups with a successful look-ahead of ten years are quite rare! In retrospect we must rate FORTRAN as a successful coding technique, but with very few effective aids to conception, aids which are now so urgently needed that time has come to consider it out of date. The sooner we can forget that FORTRAN has ever existed, the better, for as a vehicle of thought it is no longer adequate: it wastes our brainpower, is too risky and therefore too expensive to use. FORTRAN's tragic fate has been its wide acceptance, mentally chaining thousands and thousands of programmers to our past mistakes. I pray daily that more of my fellow-programmers may find the means of freeing themselves from the curse of compatibility.*
De vader van Fortran, John Backus, hield zich nog voordat Fortan 66 uitkwam vanaf eind jaren vijftig alweer bezig met een nieuwere taal, *Algol*, waar later ook op het CWI aan gewerkt is.

De evolutie van de Fortran standaard is met veel politiek geharrewar gepaard gegaan. Bij overgang van Fortran 77 naar Fortran 90 is na vele vergaderingen besloten om Fortran 90 geheel backwards compatible te maken met Fortran 77. Wel werden een aantal features van Fortran 77 *obsolescent* genoemd - een sterk advies om deze niet meer te gebruiken omdat er betere alternatieven beschikbaar zijn. In Fortran 95 zijn een aantal features die in Fortran 90 *obsolescent* waren wel geheel verwijderd. Elke Fortran standaard beschrijving bevat een hoofdstuk waarin de *deleted* en *obsolescent* features worden beschreven. De oude *fixed source form* is in versie Fortran 2003 nog steeds toegestaan, al staat dit formaat al sinds Fortran 95 bij de obsolescent features vermeld. Over de manier waarop een standaard tot stand komt en hoe standaardisatie-commissies tot besluiten zijn gekomen is er veel te leren van Fortran. Het is niet verwonderlijk dat bedrijven als Sun, Microsoft en Borland liever het heft in eigen hand houden om sneller vooruitgang te boeken met de taalontwikkeling van resp. Java, C# en Delphi.

[1] The Humble Programmer - Edsger W. Dijkstra - ACM Turing Lecture 1972

3.3.1 Lexicale analyse en Fortran.

Het ASF+SDF framework maakt gebruik van de SGLR parsing techniek (*scannerless, generalized left-to-right rightmost-derivation parsing*). Scannerless wil zeggen dat een sglr parser geen gebruik maakt van een aparte lexical analyzer die een tekst alvast in hapklare tokens aan de parser aanlevert; de sglr parser verwerkt zelf elke letter en elk symbool van de invoertekst en bouwt op basis van SDF regels een parsetree waarin ook de gehele invoertekst is opgenomen, inclusief whitespace en commentaar. Dit gaat ten koste van snelheid maar voor coderenovatie en code-analyses is dit zeer geschikt omdat er zelfs op commentaar-niveau analyses of transformaties kunnen worden gedaan en de de originele broncode weer letterlijk uit de parsetree kan worden gehaald.

De sglr parser is scannerless, maar in de onderliggende SDF definitie van de taal wordt wel onderscheid gemaakt in lexicale grammaticaregels (vergelijkbaar met het scanner-deel van een parser) en de bekende context-vrije grammaticaregels. De contextvrije grammaticaregels kennen een bijzondere soort (nonterminal) genaamd `LAYOUT`, die impliciet tussen elke term in een productieregel wordt opgenomen. `LAYOUT` is hierbij de lexicale definitie van alle witruimte tussen taalelementen in een tekst. Voor de meeste talen is de layout-definitie zoals die wordt meegeleverd in de SDF module 'basic/whitespace' voldoende:

```
 lexical syntax
  [\ \t\n] -> LAYOUT       %% space, tab and newline are whitespace
context-free restrictions
  LAYOUT? -/- [\ \t\n]     %% and are treated as a whole
```

Voor regel-georienteerde talen als Fortran kan het nodig zijn om hiervan af te wijken omdat er onderscheid moet kunnen worden gemaakt tussen de verschillende Fortran programmaregels. Het newline-token wordt dan opgenomen in de grammaticaregels. Ook de kolom-indeling die (fixed source form) Fortran kent, met de speciale betekenis van posities 1 en 6, vormen een probleem bij de lexicale syntaxdefinitie omdat er geen mogelijkheid is om kolomposities als voorwaarde of (lexicaal) element aan te geven.

Nog een aspect dat Fortran een lastige taal voor scanners/parsers maakt is dat spaties eigenlijk helemaal niet significant zijn, behalve als ze in strings staan. Een populair voorbeeld, beschreven in [1] luidt ongeveer als volgt:

```
DO 5 I = 1,25       ! DO-loop with label 5 and I running  1-25
DO5I = 1.25         ! assignment of value 1.25 to variable DO5I
DO5I = 1,25         ! DO-loop with label 5 and I running  1-25
DO 5 I = 1.25       ! assignment of value 1.25 to variable DO5I
D O 5 I = 1 . 2 5   ! assignment of value 1.25 to variable DO5I
D O 5 I = 1 , 2 5   ! DO-loop with label 5 and I running  1-25
```

Pas op het moment dat de komma is gelezen kan de parser beslissen dat het hier om een DO-loop gaat en niet om een assignment statement. De SGLR parser heeft hier geen problemen mee omdat deze redelijk ver vooruit kan kijken (lookahead theoretisch onbeperkt). De laatste twee regels zijn vrij vergezocht maar geldige Fortran *fixed source form* code en lastiger in SDF regels te vatten. In het *free source form* formaat is het gelukkig niet toegestaan om willekeurig spaties tussen 'tokens' te plaatsen al accepteren veel compilers dit nog wel.

In par. 5.1.3 is beschreven welke oplossingen hiervoor gevonden zijn.

Hoofdstuk 4

Onderzoeksmethode

Het maken van een Fortran SDF definitie kunnen we vergelijken met een software ontwikkelingstraject met de bijbehorende globale aanpak:

1. Requirements analyse. Wat zijn de eisen aan het eindproduct?
2. Constructie en implementatie.
3. Testen en validatie

De onderzoeksvraag richt zich op de kwaliteit en bruikbaarheid van een Fortran grammatica die eerst nog dient te worden gemaakt. De kwaliteit hangt samen met de requirements die aan het product gesteld worden. De requirements waren in eerste instantie kort en duidelijk: zoveel mogelijk Fortran code kunnen parsen en daar analyses op kunnen doen. Bij aanvang van het project was nog onduidelijk of Fortran als taal niet te complex was voor de Meta-Environment; dat er nog geen SDF grammatica beschikbaar was voor zo'n oude taal was al opmerkelijk.

Over het proces van grammatica-extractie van andere talen zoals Cobol is literatuur beschikbaar In het artikel *Semi-automatic Grammar Recovery* [6] beschrijven Ralf Lämmel en Chris Verhoef een aanpak om semi-automatisch een grammatica uit beschikbare bronnen te extraheren. De beschreven aanpak lijkt bij uitstek geschikt om in dit project toe te passen. De volgende stappen worden genoemd:

1. *Raw grammar extraction from a language reference, a compiler or another artifact.* Hierbij wordt bij language reference uiteraard direct aan

de Fortran standaard documenten gedacht.

2. *Resolution of static errors such as unconnected nonterminals, also called sort names, if the grammar is extracted from a nonexecutable source.*
   De Meta-Environment levert waaarschuwingen bij ongedefinieerde of ongebruikte soortnamen. Parse errors en dit soort warnings moeten worden verholpen.

3. *Extraction or definition of lexical syntax*
   Het is interessant dat dit apart genoemd wordt, en voor Fortran is dit ook een lastig probleem (zie 3.3).

4. *Test-driven correction and completion of the raw grammar if necessary*
   Pas als zowel het lexicale als contextvrije deel er is kan er worden getest met het parsen van bestaande en nieuwe Fortran testcode. Een beschikbare Fortran compiler kan in twijfelgevallen uitsluitsel geven over de geldigheid van code. GFortran (Gnu Fortran) is een bekende open source Fortran compiler.

5. *Beautification*
   Er dient een nette overzichtelijke layout voor de SDF code te worden gekozen waarbij de ISO standaard een leidraad kan zijn.

6. *Modularization*
   Is de structuur van Fortran zodanig dat het in relatief onafhankelijke losgekoppelde modulen kan worden ingedeeld? Dit maakt eventuele herbruikbaarheid van diverse modulen mogelijk en maakt de taal ook overzichtelijker.

7. *Disambiguation if necessary*
   De ASF+SDF Meta-Environment kan zonodig omgaan met ambiguïteiten in een taaldefinitie. De vraag is of het bij Fortran nodig is om dubbelzinnigheden in de taal toe te laten.

8. *Generation of a browsable version of the grammar if needed*
   Een grammatica in Html-vorm met elke nonterminals als links werkt zeer prettig en scheelt zoeken en scrollen. Het is niet noodzakelijk voor dit project.

9. *Adaptation of the grammar for the intended purpose (e.g., renovation).*
   Aangezien de use-case voor de grammatica niet bekend is zal dit door

toekomstige gebruikers van de Meta-Environment worden gedaan zodra deze een Fortran renovatie-, transformatie- of analyse-taak dienen te volbrengen.

Deze aanpak zal zoveel mogelijk gevolgd worden.

Punt twee en vier in dit stappenplan voorzien reeds in een basale vorm van testen, al is nog niet duidelijk wat de testcriteria hierbij zijn. Voor de Fortran extractie casus is de bruikbaarheid in de Meta-Environment een belangrijk criterium; met ASF regels dient Fortran code te kunnen worden geanalyseerd. Dit zal dan ook aangetoond moeten worden. Met de ISO standaard in de hand als enige alom erkende correcte Fortran definitie zal worden getracht om de SDF definitie zoveel mogelijk op de ISO standaard af te stemmen in de hoop dat daarmee de kwaliteit van de grammatica ook gegarandeerd is. Met behulp van een rule coverage meting zal worden bepaald of de gebruikte Fortran testcode wel voldoende gevarieerd is om alle aspecten van de Fortran grammatica te kunnen testen op 'parseerbaarheid'.

Hoofdstuk 5

Onderzoek

Zoals in hoofdstuk 4 vermeld zullen eerst de grammar recovery stappen moeten worden gevolgd voordat er iets met de grammatica gedaan kan worden. Elke stap wordt beschreven. Daarna volgt een test met gebruik van ASF transformatieregels en een coverage analyse van de Fortran code die als testdata voor de grammatica is gebruikt.

5.1 Grammar Recovery

In hoofdstuk 4 is het grammar recovery proces in negen stappen beschreven. De eerste zes worden in de volgende hoofdstukken uitgewerkt. Stap 7, *disambiguation* bleek vooral bij de lexicale definitie (stap 3) en de daaropvolgende testfase al nodig te zijn en is niet apart beschreven, ook al heeft het zeer veel tijd gekost om ambiguïteiten in de grammatica op te lossen.

Stap 9, het aanpassen van de grammatica voor een specifieke use-case blijft ook buiten beschouwing aangezien dat buiten de scope van de ontwikkeling van de baseline grammar ligt.

5.1.1 Grammar extraction (stap 1)

Een taal met zo'n lange geschiedenis brengt voor de extractie van een SDF grammatica extra problemen met zich mee, zoals:

- Welke versie(s) van de standaard(en) kunnen worden gebruikt?

- Wat is de kwaliteit van de standaard?
- Kan de grammatica uit de standaard geextraheerd worden?
- Kan de extractie makkelijker uit een compiler of iets dergelijks gedaan worden?
- Kan de SDF grammatica de standaard wel volledig beschrijven?

Een zeer praktische vraag voor dit project is echter: hoe kom je aan een Fortran standaard? De *International Organization for Standardization* (ISO) beheert de standaard documenten en biedt deze voor CHF342,- (circa 207 euro) aan in zowel pdf, cdrom als papieren vorm[1]. Draft-versies van deze documenten slingeren nog wel rond op het Internet maar zijn die wel te vertrouwen? De CWI-bibliotheek bezit een papieren versie van de Fortran 90 standaard (officiele code: ISO/IEC 1539:1991(E)). Op enkele uitzonderingen na is deze standaard compatible met FORTRAN 77 in de zin dat *With limitations noted in 1.4.1, the syntax and semantics of the International Standard commonly known as 'FORTRAN 77' are contained entirely within this International Standard* [3]. Twee vliegen in een klap. Zoals eerder beschreven zijn in Fortran 90 nog geen bestaande features van eerdere versies van Fortran 'deleted' verklaard.

Het document *'ISO/IEC 1539 second edition, 1991-07-01'* beschrijft de taal Fortran zo uitgebreid en nauwkeurig mogelijk. Het is 396 pagina's lang en bevat 13 hoofdstukken die elk een bepaald aspect van de taal beschrijven. Voor de taalbeschrijving wordt een vorm van BNF gebruikt (Backus bedacht dit formaat overigens om Algol mee te beschijven). Met Fortran voorbeeldcode worden diverse constructies in de standaard toegelicht. Ter vergelijking: een van de bekendste boeken over Fortran, *Fortran 90/95 explained* [8] beschrijft deze taal in 340 pagina's. Voor de grammatica-extractie is dit laatste boek echter ongeschikt omdat het geschreven is voor een geheel andere doelgroep, namelijk zij die de taal Fortran willen leren om daarmee te programmeren. Voor het maken van een Fortran parser met SDF is het niet nodig om in Fortran te kunnen programmeren; er is kennis van de taal Fortran op grammatica-niveau nodig. Voor het maken van tests is basale syntax-kennis handig, maar kennis van bijvoorbeeld de ingebouwde functies en procedures van Fortran is niet nodig.

Het belangrijkste deel van de standaard is Annex D, de syntax regels. Deze bijlage is een verzameling van alle Fortran productieregels die in de diverse

[1] http://www.iso.org

hoofdstukken toegelicht worden. Deze verzameling zou een kandidaat kunnen zijn voor (semi-)automatische grammatica extractie, ware het niet dat deze niet in digitale vorm beschikbaar was.

Wel als download (gratis) beschikbaar is een BNF grammatica die vrijwel letterlijk de Fortran 77 én Fortran 90 standaard naamgeving en nummering volgt en die gebruikt wordt in een framework voor compiler constructie, het ELI framework [2] Door dit als uitgangspunt te nemen is vrijwel zeker dat er Fortran code kan worden herkend omdat het al in bestaande ELI projecten is toegepast. Stap 1-3 van de semi-automatic grammar recovery kon nu worden toegepast met als gevolg dat al snel ruim 1700 regels BNF in SDF werden omgezet. Dit is 'semi-automatisch' gedaan door reguliere expressies te gebruiken in de VIM editor, in combinatie met handmatige controle. Binnen enkele dagen was de SDF grammatica vrij van SDF parse-errors. Dit wil echter nog niet zeggen dat er ook Fortran code kan worden gelezen: de lexicale syntax zou geheel handmatig moeten worden gemaakt omdat dit in een traditionele omgeving als ELI aan een aparte scanner wordt overgelaten. Verder dient de grammatica nog te worden getest en ge-refactored naar een vorm die zo dicht mogelijk bij de ISO standaard ligt. De ASF+SDF omgeving met sglr parsing maakt het ook mogelijk om meerdere (BNF-)productieregels op een kortere, duidelijkere of meer op de standaard gelijkende vorm te beschrijven. Dat laatste heeft de voorkeur, zolang het niet tot gevolg heeft dat de grammatica niet meer bruikbaar is ten gevolge van parse errors of ambiguïteiten.

5.1.2 Resolution of static errors

De ELI Fortran grammatica ligt dicht tegen de ISO standaarddefinitie aan maar heeft toch nog diverse bewerkingen nodig voordat het als SDF bruikbaar is en meer lijkt op de standaarddefinitie. Dit komt vooral omdat de ELI versie LALR(1) moet zijn en geen EBNF notatie kent. Door de standaard te vergelijken met de ELI regels blijkt ook dat niet elke constructie correct is, en moeten er termen worden toegevoegd. Ook zijn er vele regels die in SDF slechts een regel nodig hebben. Zo kan het gebeuren dat twee regels in de ELI grammar in SDF uiteindelijk een aangepaste regel wordt die vrijwel identiek is aan de originele ISO regel. De drie versies onder elkaar:

```
Originele regel uit de ISO standaard:
R804 else-if-stmt is ELSE IF (scalar-locical-expr)
```

[2] Toolset for Compiler Construction; http://eli-project.sourceforge.net

```
                                THEN [ if-construct-name ]

%ELI voorbeeld:
xElseIfStmt:
 xLblDef 'else' 'if' '(' xExpr ')' 'then' xEOS .
xElseIfStmt:
 xLblDef 'elseif' '(' @<Logical Expression@> ')' 'then' xEOS .

%%uiteindelijke SDF regel:
%%R804
 LblDef 'else' 'if' '(' ScalarLogicalExpr ')' 'then'
                        IfConstructName? EOS -> ElseIfStmt
```

De terminals 'else' en 'if' mogen in Fortran ook aan elkaar geschreven worden als 'elseif'. Aangezien de default LAYOUT tussen elke term in een SDF (contextvrije) regel optioneel is, is het niet nodig om 'elseif' in een aparte SDF regel op te nemen. Dit zou zelfs ambigu worden. Wat nu wel direct opvalt is dat er geen spaties tussen de letters van 'else' mogen staan, anders had het als 'E' 'L' 'S' 'E' moeten worden geschreven (of in kleine letters, maar single quotes in SDF impliceert hoofd+kleine letters). Door dit niet te doen is de grammatica wel een stuk leesbaarder en er is geen Fortran leerboek waarin zal worden aanbevolen om spaties tussen Fortran 'keywords' of getalconstanten te plaatsen. Een google code search op `lang:fortran "e l s e"` levert welgeteld een enkele hit. Voor dit soort uitzonderingen is een kleine code-aanpassing aantrekkelijkere oplossing dan het aanpassen van de SDF regels. Om ook spaties in identifiers en getallen toe te laten zou de lexicale definitie hopeloos in conflict komen met de LAYOUT definitie. Ook hierbij geldt dat er geen serieuze Fortran code is waarbij spaties tussen de cijfers van getallen wordt geschreven, ook niet in de *fixed source form* waarbij dit is toegestaan. Een workaround voor de uitzonderingen is om de code eerst door een preprocessor te halen die alle (overbodig) spaties uit de code verwijderd om het daarna te parsen op basis van een leesbare SDF grammatica.

In de ELI grammar staan talloze regels die alleen maar doorverwijzen naar de lexical analyzer waarvan de werking niet duidelijk is, zoals de `@<Logical Expression@>` constructie in het voorgaande voorbeeld. Zonder aanpassingen levert dit direct een parse-error op in de Meta-Environment. Deze fouten kunnen worden verholpen door de standaardversie van de betreffende soort te volgen. De lexicale definitie van Fortran voor ELI zit verweven in de C code van de scanner. De lexicale syntaxdefinitie is echter wel in de Fortran standaard

beschreven en is handmatig overgenomen in de SDF lexicale definitie. De specifieke Fortran code opmaak levert hier wel een probleem op.

5.1.3 Lexical syntax definition: fixed en free source form

Velen kennen Fortran als de taal waarbij je op de 7e positie op elke regel moet beginnen. Dit heet nu de (afgeraden) fixed source form. Nog steeds wordt aanbevolen om de eerste 6 plaatsen vrij te houden om code leesbaar te houden voor oudere compilers. Er is een Fortran programma beschikbaar om Fortran programma's van de Fortran 77 fixed source form naar de Fortran 90 free source form te converteren [8]. Toch is er nog heel veel Fortran code in de oude vorm te vinden. In de fixed form zijn de volgende regels van belang:

1. Een `C`, `c` of `*` in de eerste positie betekent dat de betreffende regel commentaar is
2. Posities 1 t/m 5 zijn bestemd voor een numeriek label van 1-99999
3. Een willekeurig non-blank teken op positie 6 betekent continuatie van de vorige regel (ook een `;` of `!`)
4. Sourcelines gaan tot en met positie 72
5. Een `!` op elke positie behalve positie 6 begint een comment dat loopt tot het einde van de regel
6. Statements mogen worden gescheiden door een of meer puntkomma's.

Om in de lexicale SDF definitie met het concept 'regel' te kunnen werken is besloten om het newline teken (`\n`) een prominente rol toe te kennen als end-of-statement en end-of-comment markering. Het grote voordeel hiervan is dat de Fortran code geen voorbewerkingen hoeft te ondergaan om bijvoorbeeld het begin van elke regel herkenbaar te maken met een unieke "magische" code. De aanpak met voorbewerking is in eerste instantie wel uitgeprobeerd en werkt, maar om voorbewerkingen zoveel mogelijk te voorkomen is hiervan afgezien. Een derde mogelijkheid zou zijn om de sglr parser aan te passen en 'begin-van-regel' of 'kolompositie x' binnen SDF als lexicaal soort te definieren waarop de parser kan besluiten om een regel wel of niet te accepteren.
Uitgaande van de eerste oplossing met gebruik van het newline-teken is de

definitie van commentaarregels die beginnen met een `C`, `c` of asterisk in de eerste positie nu als volgt (hierbij is EOS het end-of-statement soort):

```
%% Fixed source form comment rules
 ("\n" [Cc\*] ~[\n]*)+       -> FixedComment
 FixedComment "\n" [\ \t]*   -> EOS
```

Hieruit blijkt dat de newline van de 'vorige' regel dus gebruikt wordt om te bepalen of een teken op de 1e positie van een regel staat. Deze aanpak werkt, met de beperking dat een laatste programmaregel wel altijd met een newline dient te worden afgesloten en dat een programma niet direct met een dergelijke commentregel mag beginnen. Als je nu echter kolomposities net als newlines en tabs zou kunnen beschouwen als onzichtbare lexicale tekens zijn deze beperkingen niet nodig en zou de eerste regel bijvoorbeeld kunnen luiden:

```
 (pos(1) [Cc\*] ~[\n]* "\n")+ -> FixedComment
```

`pos(1)` is hier een voor de scanner een aanduiding dat de huidige scanpositie 1 dient te zijn om de regel verder te mogen parseren. Omdat dit een aanpassing van de parser vergt is deze oplossing niet verder uitgewerkt. Het is wel een aandachtspunt voor de verdere ontwikkeling van de Meta-Environment als het in staat moet zijn om ook talen als Python en ABC moet kunnen parsen. Bij deze talen is het inspringen (*indent*) en terugspringen (*dindent*) vanaf een kantlijn een belangrijk gegeven wat vertaald moet worden naar 'indent' en 'dindent' tokens, vergelijkbaar met de *begin* en *end* tokens bij talen als Algol en Pascal. Ook hier zou uiteraard een pre-processing fase de indent en dintent tokens wel zichtbaar kunnen maken, maar de sglr parser heet niet voor niets *scannerless* en zou dit dus liefst zonder aparte scanner-fase moeten kunnen.

De *lexicale* SDF code voor Fortran die uiteindelijk is geschreven is niet bijzonder groot maar heeft tijdens de ontwikkeling veel problemen opgeleverd. Om free source form continuatietekens te kunnen verwerken is bijvoorbeeld een behoorlijk complexe regel gebruikt:

```
 "&" [\ \t]* ("!" ~[\n]*)? "\n" ([\ \t]+ "&")? -> Continuation
 Continuation -> LAYOUT
```

Dit is noodzakelijk om de volgende Fortran code zonder pre-processing-stap te kunnen lezen:

```
  IF( LABEL .NE. 0 .AND.                                              &
& LCHAR .GE. 11 .AND.(BUFFER(:7) .EQ. 'FORMAT(' .OR.                  &
&                     BUFFER(:7) .EQ. 'format(') .AND.                &
  BUFFER(LCHAR-2:LCHAR-2) .EQ. ')') THEN
   IF (LEN-LENST.GE.2) STAMNT(LENST+1:LENST+2) = '  '
     LENST = MIN(LENST+2, LEN)
     GO TO 99
   ENDIF
  IF (OVFLW) WRITE(*,  '(" At least one statement was too long to h&
  &ave a necessary blank added")')
```

De stukken whitespace tussen de ampersand tekens zijn nu `LAYOUT` geworden en worden aldus 'verweven' in de contextvrij grammatica definitie van bijvoorbeeld een expressie of IF statement zonder betekenis te krijgen. Bij de laatste twee regels worden de `&` tekens echter als onderdeel van de string gezien omdat de definitie van de string geen rekening kan houden met wat er allemaal in de string zelf is opgenomen:
`[\'] (~[\'] | "''")*  [\'] -> SconSingle`
is de definitie van een stringconstante die door single quotes wordt omsloten, met als uitzondering de quote zelf die in Fortran door een extra quote mag worden *escaped*. De tekst `It's a nice day` wordt in een Fortran string dus genoteerd als `'It''s a nice day'`.

Het oplossen van lexicale ambiguiteiten is bijzonder lastig gebleken maar het is dus mogelijk gebleken om Fortran programma's te kunnenlezen zonder gebruik te maken van een aparte pre-processing fase. Dit is gelukt zonder *reject* of *prefer* of *avoid* attributen te gebruiken. De lookahead-restrictie is wel veelvuldig toegepast. Om in bovenstaand voorbeeld te voorkomen dat LAYOUT en Continuation op diverse manieren geparsed kunnen worden is het verboden om in de parsetree een ampersand na een LAYOUT productie te hebben. Dit dwingt de parser om het ampsersand onderdeel te laten zijn van een eerdere afleiding van LAYOUT. Het is ook te lezen als: "spaties, tabs en ampersands zoveel mogelijk bij elkaar houden":

```
  LAYOUT? -/- [\ \t\&]
```

De Meta-Environment helpt bij het oplossen van ambiguïteiten door alle mogelijke dubbele afleidingen in de grafisch weergegeven parsetree te laten zien maar kent geen *fully automated lexical disambiguation* [9].

5.1.4 Test-driven correction

De eenvoudigste manier om een Fortran grammatica te testen is om met behulp van de Meta-Environment steeds grotere stukken Fortran code te proberen te parsen en parse-errors te verhelpen door de grammatica te verbeteren. Als het parsen lukt kan er door de parsetree van de Fortran code worden gewandeld om bijvoorbeeld te controleren of een IF wel als `IfStmt` wordt herkend en niet als `Comment`. In veel gevallen moet vervolgens de Fortran standaard geraadpleegd worden als blijkt dat de ELI grammar die als bron diende afwijkt van de standaard. De fouten in de ELI Fortran grammatica zijn in de SDF grammatica verbeterd.

Het vinden van goede testcode blijkt een probleem. Miljoenen regels Fortran zijn geschreven door 'netjes' te schrijven, maar juist de uitzonderingsgevallen brengen fouten in de grammatica aan het licht. Zo is er maar erg weinig code te vinden waarbij een regel wordt afgesloten met meerdere puntkomma's terwijl dit volgens de standaard is toegestaan. Een aanpassing in de lexicale definitie hiervoor is nog zeer recent toegevoegd. In de ruim 400000 regels Fortran code die eerder als test werden gebruikt was een puntkomma niet als end-of-statement aan het eind van een regel gebruikt. Het is ook overbodig en waarschijnlijk een vergissing van een programmeur die C of Pascal gewend is.

Door diverse Fortran testprogramma's in enkele vensters van de Meta-Environment open te laten staan tijdens het wijzigen van de grammatica kan snel worden geconstateerd of er een verbetering (meer Fortran code wordt herkend) of juist een fout (minder Fortran code wordt herkend) in de grammatica wordt aangebracht. Tijdens deze fase worden diverse beperkingen in de Meta-Environment geconstateerd. Zo is geldige Franstalige Fortran code correct te parsen maar loopt de syntax highlighting hopeloos uit de pas met de tekst en kunnen bestanden groter dan 1 MByte niet worden geopend. Voor grotere tests is het noodzakelijk om de onderliggende tools van de Meta-Environment vanuit een script te runnen. In eerste instantie zal dan alleen getest worden of (veel) Fortran code kan worden herkend. Dit is zoals gezegd nog geen garantie dat de grammatica compleet of bruikbaar is.

Om een Fortran testprogramma door de parser in een parsetree om te laten zetten zijn slechts twee tools uit de Meta-Environment van belang:

```
pt-dump -m Fortran90 -o fortran.tbl
```

Dit commando zal de SDF grammatica omzetten in een parsetable met filenaam fortran.tbl waarmee een parser Fortran code kan gaan lezen. Zolang de SDF broncode niet wijzigt kan deze parsetable bewaard worden voor de sglr parser. De parsetable bevat alle lexicale en contextvrije productieregels van de grammatica. Met

```
dump-productions  fortran.tbl
```

kunnen deze productieregels getoond worden. Dit zal later gebruikt worden voor de coverage-test.

De sglr parser kan een Fortran bestand met de naam 'bigtest.trm' gaan inlezen en converteren naar een parsetree die we kennen uit de Meta-Environment waar deze grafisch kan worden weergegeven (als het bestand kleiner is dan 1MByte). De parser wordt als volgt gestart:

```
sglr -A  -p fortran.tbl -i bigtest.trm  -fi -fe -o bigtest.pt
```

De -A optie zorgt ervoor dat eventuele ambiguïteiten als foutenmeldingen zullen worden gemeld. De Fortran grammatica is echter niet ambigu. De sglr parser doet er ongeveer 4 minuten over om 400.000 regels Fortran te parsen.

5.1.5 Beautification

Hoe ziet een mooie grammatica eruit? In *"Coding standards for The Meta Environment"* [3] worden een aantal richtlijnen voor overzichtelijke SDF code beschreven:

1. *Group productions on the non-terminal on the right-hand side of productions*
   De ELI bnf code waarvan de SDF code is afgeleid was een mix van Fortran77 en Fortran90 regels die elkaar deels overlapten maar verspreid over een enkel bestand waren geplaatst waarbij de producties waren gegroepeerd op basis van de Fortran 77 en de latere Fortran 90 regels. Dit was niet overzichtelijk en aangezien de Fortran 77 standaard niet beschikbaar was is besloten alles zoveel mogelijk volgens de Fortran 90 standaard te beschrijven waarvan Fortran 77 immers een subset is. Door deze uniforme aanpak konden worden voldaan aan de richtlijn om producties op soort te sorteren. Het nadeel is dat voor een eventuele ontvlechting naar Fortran 77 en Fortran 90 standaard er opnieuw moet worden bepaalt wat de Fortran 77 regels zijn. Hier-

[3] http://www.cwi.nl/htbin/sen1/twiki/bin/view/Meta-Environment/Documentation

voor is de beschikking over een Fortran 77 standaard document echter onontbeerlijk.

2. *Do not duplicate productions between priority sections and normal syntax sections*
   De Fortran expressie syntax is in de standaard zeer uitgebreid beschreven met productieregels waarin de prioriteit reeds met meerdere niveaus is beschreven. Hoewel dit bij expressies een zeer 'diepe' parsetree oplevert is hier de keuze gemaakt om zo dicht mogelijk de standaard te volgen. Het herschrijven van de expressiesyntax met de SDF prioriteit-syntax is een oefening die aan de lezer wordt overgelaten.

3. *Fix all warnings mentioned by the SDF checker*
   Ongebruikte soortnamen, ongedefinieerde soortnamen, deze waarschuwingen zijn eenvoudig te verhelpen. Zie ook hfdstk 5.1.6.

4. *Non-terminals (sorts) start with a capital letter, and continue with javaCase*
   De Fortran standaard gebruikt deze notatievorm niet en schrijft nonterminals in lowecase met streepjes ertussen. De ELI syntax leek meer op de javaCase maar begon niet met hoofdletters. Er is gekozen om Fortran standaardnamen als *main-program* en *block-data-stmt* te schrijven in javaCase als *MainProgram* en *BlockDataStmt*. Door de nummering van de producties in de Fortan standaard als commentaar bij de SDF regels te plaatsen is de SDF code goed te vergelijken met het standaard document.

Wat bij deze opsomming niet wordt genoemd is het al of niet gebruiken van de EBNF notatie. In de gebruikte ELI code waren de producties in BNF als een LALR(1) grammatica opgesteld. In SDF is veel meer vrijheid in de (EBNF) notatie en kunnen veel regels in de oude ELI code worden herschreven tot een enkele SDF regel. Dit kan vooral bij lijsten en optionele elementen in productie een flinke vereenvoudiging betekenen. Lastiger is het om te bepalen of de twee vormen wel volledig uitwisselbaar zijn. Er is gekozen om de SDF code zoveel mogelijk in overeenstemming te laten zijn met de standaard, waar optionele elementen en lijst-notatie ook wordt gebruikt (zij het in een eigen formaat). Het refactoren en verkorten van de ELI/BNF naar SDF is grotendeels handmatig gedaan, waarbij als kwaliteitscontrole het foutloos en zonder ambiguiteiten kunnen blijven parsen van de Fortran testcode is gebruikt. Hieronder volgt een voorbeeld van oud naar nieuw,

waarbij een hulpsoort overbodig is gemaakt. Na de grammatica extractie zag regel R432 er als volg uit:

```
Expr                          -> AcValueList
AcValueList1                  -> AcValueList
Expr ',' Expr                 -> AcValueList1
Expr ',' AcImpliedDo          -> AcValueList1
AcImpliedDo                   -> AcValueList1
AcValueList1 ',' Expr         -> AcValueList1
AcValueList1 ',' AcImpliedDo -> AcValueList1
```

Door gebruikt te maken van de lijstnotatie in SDF wordt dit een stuk eenvoudiger:

```
Expr                   -> AcValue
AcImpliedDo            -> AcValue
{AcValue ","}+         -> AcValueList
```

Deze vereenvoudiging levert uiteraard problemen op als andere productieregels gebruik maken van AcValueList1. De twee voorlaatste regels in dit voorbeeld kunnen met de SDF 'or' operator ook in een regel worden geschreven:

```
Expr | AcImpliedDo  -> AcValue
```

Toch is hier meestal de voorkeur naar twee of meer losse regels gegaan omdat dat er netter uitziet.

De resulterende SDF code ziet er zeer overzichtelijk uit en met de Fortran standaard ernaast is gemakkelijk te zien waar het hiermee overeenkomt en waar het afwijkt.

5.1.6 Modularization

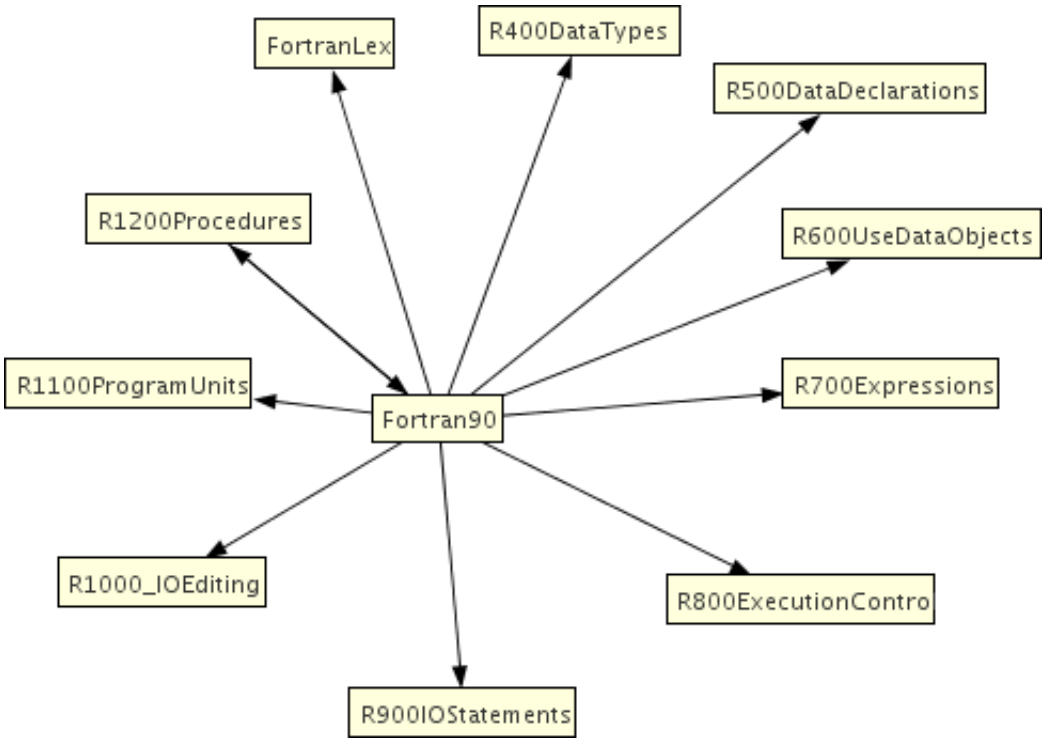
Het meeste werk aan de grammatica is gedaan in een enkel SDF bestand van ongeveer 1656 regels en 760 producties. Hoe kunnen deze het best over meerdere modulen worden verdeeld? Is er een compositie-architectuur in Fortran te ontdekken? Om hier het antwoord op te vinden hoeft eigenlijk alleen maar naar de inhoudsopgave van de standaard te worden gekeken. In de standaard worden de Fortran producties keurig naar hoofdstuk genummerd en verdeeld over de volgende hoofdstukken:

- Fortran terms and concepts (sectie 2, regels R201-R216)
- Characters, lexical tokens and source form (sectie 3, regels R301-R313)
- Intrinsic and derived data types (R401-R435)
- Data object declarations and specifications (R501-R549)
- Use of data objects (R601-R631)
- Expressions and assignment (R701-R740)
- Execution control (R801-R844)
- Input/Output statements (R901-R924)
- Input/Output editing (R1001-R1017)
- Program units (R1101-R1112)
- Procedures (R1201-R1226)

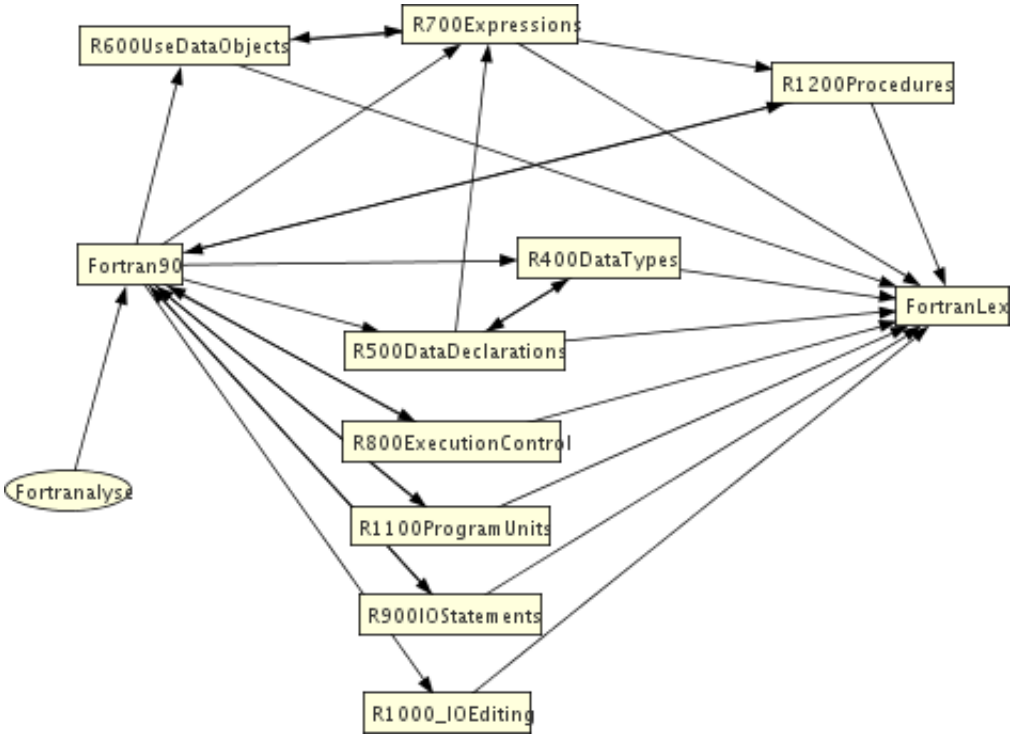
Ook hier is gekozen om niet van de standaard indeling af te wijken. De secties 4 t/m 12 zijn allen in aparte SDF files met overeenkomstige naam geplaatst. sectie 3 is FortranLex.sdf genoemd en sectie 2 is de hoofdmodule geworden die vrijwel alle overige modulen importeert. Deze heet Fortran90.sdf.

In de Meta-Environment ziet de import graph er als volgt uit: (zie fig 5.1).

Wat opvalt is dat het lijkt alsof de verschillende modulen totaal onafhankelijk zijn van elkaar. Dat is echter niet zo. De dependencies tussen de verschillende modulen waren nog niet goed uitgezocht en door een van de modulen de topmodule te laten importeren (in dit geval importeert R1200Procedures de centrale Fortran90 module) klaagt de Meta-Environment (versie RC2.0 althans) niet over onbekende soortnamen. Aangezien hiermee ook waarschuwingen van de SDF checker worden omzeild is dit niet de gewenste eindsituatie. Latere versies van de SDFchecker van de Meta-Environment gaven meer waarschuwingen en het her-modulariseren is een vorm van refactoring wat nog gaande is in de vorm van ambachtelijk handwerk. De volgende figuur toont een iets realistischer import graph (fig 5.2).



Figuur 5.1: Fortran import graph (Meta-environment Radial view)



Figuur 5.2: Fortran import graph met juiste dependencies

5.1.7 Generating browsable version

Met de juiste tools zou het niet moeilijk moeten zijn om een web-browsable versie van de Fortran SDF grammatica te genereren. Voor diverse talen is dit al gedaan, zie [4].

SDF-browsing is een feature die in de Meta-Environment zou kunnen worden ingebouwd om snel naar de definitie of het gebruik van een soort te kunnen zoeken, ongeveer zoals dat met IDE's als Visual Studio en Eclipse ook kan. Een export naar Html functie zou uiteraard ook in de Meta-Environment kunnen worden ingebouwd.

5.2 Testen van de grammatica

5.2.1 Gebruik van ASF transformatie

Is deze Fortran grammatica bruikbaar? Dat is afhankelijk van het doel, de use-case. Aangezien met de ASF+SDF Meta-Environment meestal ASF transformatieregels worden gebruikt ligt het voor de hand om te testen of hiermee nu ook Fortran code kan worden bewerkt. Dit blijkt het geval, zij het met een speciale 'gebruiksaanwijzing' voor de ASF syntax. Het equation-deel van een ASF definitie moet namelijk voldoen aan de (lexicale en contextvrije) definitie van Fortran. Hierin is nu echter een speciale rol toebedeeld aan het newline teken, waardoor ASF regels niet zomaar onder elkaar mogen worden geplaatst. Door nu gebruik te maken van Fortran line continuation characters (`&`) in de ASF regels kunnen deze toch worden opgesplitst in meerdere regels. Deze afwijkende ASF-syntax maakt deze ASF-regels wel zeer Fortran-specifiek, maar dat zijn Fortran transformatieregels uiteraard altijd. Een simpel voorbeeld dat IF-THEN-ELSE-ENDIF statements herschrijft naar IF-THEN-stop-ENDIF ziet er als volgt uit (gebaseerd op producieregel R802 uit de standaard):

```
equations  &
 [R4] IfTS1 EPConstruct*1 ElseStmt1 EPConstruct*2 EndIfStmt1 =  &
                   IfTS1              stop
                   EndIfStmt1
```

De & tekens zijn hier nodig op plaatsen waar Fortran geen newline verwacht maar waar je de ASF regels wel graag van elkaar wilt scheiden voor de

[4] http://www.cs.vu.nl/grammars/help.html

duidelijkheid. Anderzijds moet er na het stop statement echt een newline komen omdat Fortran statements gescheiden moeten worden door newline.

Er zijn nog enkele transformaties met behulp van ASF getest en geslaagd. Hierbij is opgevallen dat het niet mogelijk is om van DO-LOOPS de 'body' te kunnen herkennen. Dit is een probleem dat stamt uit Fortran 77 syntax, waarbij een DO niet gevolgd wordt door een END DO maar middels een label wordt aangegeven tot waar de lus loopt. Door een aanpassing in de grammatica is het echter wel mogelijk om van pure Fortran 90 code waarin de Fortran 77 constructie niet wordt gebruikt de structuur van DO-LOOPS te herkennen. Dit is een voorbeeld van het aanpassen van de baseline grammar tot een *grammar for the intended purpose*.

5.2.2 Kwaliteit en testkwaliteit

In par. 3.2 is iets gezegd over de kwaliteit van een grammatica. De Meta-Environment ondersteunt de ontwikkelaar van een grammatica met waarschuwingen over ongedefinieerde en ongebruikte soorten, maar geeft niet aan of een productieregel overbodig is. Door nu een coverage-test uit te voeren op een grote hoeveelheid testdata kan worden bepaald welke productieregels nader bekeken moeten worden. De parsetree die door de parser wordt geproduceerd uit de Fortran testcode bevat niet alleen de complete broncode maar ook de producties waarin de tekst is ontleed. In bijlage B is een C programma toegevoegd wat deze productieregels uit de parsetree verzamelt en telt. Vervolgens kan deze lijst worden vergeleken met de productieregels in de brongrammatica. Een eerste resultaat bij het toepassen van deze vergelijking was dat een grote Fortran bibliotheek van 410KLOC die gebruikt is als test minder verschillende productieregels gebruikte dan een ander kleiner testbestand. Achteraf was dit niet verwonderlijk aangezien het de LAPack bibliotheek betrof. Dit is een bekende verzameling Lineaire Algebra routines. Het aantal regels code in deze verzameling is geen garantie voor diversiteit in het taalgebruik; print en IO-format statements e.d. werden maar een keer gebruikt. Wat opviel was dat een Fortran77-stijl programma van slechts 3KLOC 501 productieregels benutte waar de LApack bibliotheek er 402 aanspreekt. Of hieruit de conclusie mag worden getrokken dat Fortran 90 'beter' is dan Fortran 77 is, is iets wat een ander nu kan uitzoeken die de Fortran grammatica voor verdere analyses gaat gebruiken.

Een test waarbij werkelijk alle productieregels werden benut is niet gevonden noch gemaakt. Zoals in par. 3.2 beschreven zou er code uit de brongram-

matica kunnen worden gegenereerd om deze te vergelijken met 'correcte' code. Nu is wel duidelijk dat dit hopeloos mis zal gaan aangezien er op diverse plaatsen in de grammatica is gekozen voor een vereenvoudiging of het 'relaxen' van de regels, waardoor codegeneratie zeker ongeldige Fortran zal kunnen produceren. Veel regels zijn ook niet in SDF te beschrijven en staan in de iso standaard opgesomt als talloze 'Constraint' notities zoals *'An internal subroutine must not contain an ENTRY statement'*. Zo blijft het ISO document een waardevol orakel om de geldigheid van Fortran code te bepalen.

Hoofdstuk 6

Resultaten

Het belangrijkste resultaat is een bruikbare Fortran grammatica, gebaseerd op het ISO/IEC 1539:1991(E) document. De grammatica is opgeleverd in de vorm van een verzameling van 11 SDF modulen met een totaal van 1703 regels (LOC) inclusief comments. Zie bijlage C.

Het is qua omvang de grootste grammatica uit de SDF bibliotheek geworden. Enkele statistieken:

Modulenaam	aantal producties	regels code
Fortran90	94	168
FortranLex	54	148
R400DataTypes	44	116
R500DataDeclarations	141	292
R600UseDataObjects	40	98
R700Expressions	110	234
R800ExecutionControl	52	151
R900IOStatements	108	193
R1000_IOEditing	36	72
R1100ProgramUnits	29	75
R1200Procedures	60	156
Totaal	768	1703

6.1 Coverage testresultaat

Voor het coverage-testen van de grammatica zijn enkele grote Fortran pakketten met talloze Fortran bestanden samengevoegd tot een enkel te parsen bestand en is geteld hoevaak elke productieregel uit de Fortran parsetable is toegepast op de testcode. Dit is gedaan met behulp van standaard (unix) tools als `awk`, `sort`, `vim` en vooral `uniq` – een tool om gelijke (dubbele) regels in een bestand te tellen.

Als de producties uit de Fortran parsetable worden gedumped in het outputformaat van de tool 'dump-productions' zijn het er 1777. Dit aantal is zo groot omdat elke SDF regel meerdere regels in het parsetable dump formaat oplevert. Voor vergelijking met parsetree-dumps is dit formaat echter goed te gebruiken. Ter vergelijking: de Java definitie in de SDF library telt 718 regels in dit formaat. Dit zegt iets over de omvang van de taal Fortran en geeft ook aan dat de Fortran grammatica al gebruikt kan worden om metrieken van taaldefinities te bepalen. Middels het programma uit bijlage B zijn soortgelijke productieregels uit de parsetree van de Fortran testcode gehaald en vergeleken met de 1777 Fortran regels. De Fortran code uit de LA-pack routines bestaande uit 410088 regels maakte gebruik van 402 producties. Een ander Fortran pakket bestaande uit 6330 regels gebruikte 499 verschillende productieregels. Beide pakketten samengevoegd leverde dat 575 gebruikte productieregels. In een percentage uitgedrukt ten opzichte van de 1777 productieregels van de Fortran grammatica levert dit een coverage percentage van slechts 32 procent! Door steekproeven te nemen uit de productie regels die blijkbaar in geen van de ruim 416 KLOC Fortran code voorkomt kan worden bevestigd dat vele Fortan constructies inderdaad niet in de Fortran tekst voorkomt. Fortran is blijkbaar een taal die zo rijk is aan woorden dat er vele ongebruikt kunnen blijven. Het zou interessant zijn dit met andere talen te vergelijken, maar in dit onderzoek gaat het om het bepalen en verbeteren van de kwaliteit van de Fortran grammatica en niet de taal Fortran.

Wat betekent een coverage percentage van 32 procent voor de kwaliteit van de Fortran grammatica? De kans is groot dat er nog fouten in de grammatica zitten die zich manifesteren in de vorm van parse-errors bij het lezen van een nieuw stuk Fortran code. Uitgaande van code die wel door een Fortran compiler wordt geaccepteerd zal de gebruiker moeten bepalen of het een fout in de compiler of in de SDF grammatica betreft. Afhankelijk van de use-case kan het ook zo zijn dat de SDF grammatica wordt aangepast omdat het er

helemaal niet toe doet wie gelijk heeft als het om code-analyses gaat waar code-correctheid er niet toe doet.

De gevonden niet-gebruikte taalconstructies kunnen worden gebruikt om een Fortran Google code-search te doen om de test-set te vergroten en de coverage te verhogen. Deze zoekactie leverde de Gnu Fortran testcode library op bestaande uit slechts 2650 regels Fortran broncode (inclusief commentaar) met een coverage van maarliefst 40%. Dit is hoog in vergelijking met de vorige meting over 156 maal meer regels testcode. Deze code samengevoegd met de 416 KLOC testcode die eerder was gebruikt levert een totale coverage van 44% op. De nieuwe testcode bracht vervolgens ook fouten in de grammatica aan het licht die nog niet eerder waren getest. Het nut van de coveragebepaling om de test-set scherper te krijgen en fouten te vinden is aangetoond. De tooling hiervoor zou in de toekomst ingebouwd kunnen worden in de Meta-Environment als hulpmiddel voor het testen van grammatica's en het analyseren van code.

De huidige wijze van bepaling van het absolute coverage percentage is overigens nog niet waterdicht en levert lagere waarden dan in werkelijkheid waardoor de 100% nooit gehaald kan worden. Voor een vergelijking van testsets en het vinden van nieuwe testcode is de coverage bepaling goed genoeg.

Hoofdstuk 7

Conclusie

Aan de hand van [4] werd getracht in dit project "GrammarWare Engineering"toe te passen in plaats van het meestal toegepaste "Grammarware Hacking"waarbij handmatig een grammatica wordt bewerkt. Dat is vrijwel niet gelukt. Het programmeren met SDF grammatica's blijft net als bij andere talen een kunst, en de hulpmiddelen zijn nog primitief. Qua proces en werkwijze is aangetoond dat de stappen voor grammatica-extractie zoals beschreven in [6] voldoen. De *test-driven correction* fase is hierbij na de beautification en modularization fase doorgegaan op basis van coverage onderzoek.

De *lexical syntax definition* fase heeft vrij veel tijd gekost door de specifieke opmaak van Fortran. Aangetoond is dat het ondanks de lastige opmaakregels toch mogelijk is om met de beschikbare middelen en zonder pre-processing de meeste Fortran90 code te kunnen parsen. Voor Fortran77 code is een beperkte voorbewerking nodig. Om meer talen met specifieke opmaak-regels (zoals Python of ABC) te kunnen verwerken met de Meta-Environment wordt aanbevolen om voorzieningen in de Meta-Omgeving aan te brengen waarmee betekenisvolle opmaak-kenmerken makkelijker kunnen worden herkend.

Gedurende dit project is geprobeerd de SDF code zoveel mogelijk op de beschrijving in de ISO standaard te laten lijken. De kwaliteit van de jaren oude ISO standaard met bijbehorende aanvullingen en verbeteringen is hoog. Door naamgeving, regelnummering, indeling en structuur van de SDF code zoveel mogelijk door de standaard te laten bepalen biedt de SDF grammatica houvast voor iedereen die zekerheid wil over de juiste taaldefinitie.

Ondanks dit streven naar gelijkheid met de standaard is op vele plaatsen echter afgeweken van de strenge beperkingen die de standaard voorschrijft omdat deze niet in SDF zijn uit te drukken. Semantische regels en type-controle dienen te worden toegevoegd afhankelijk van de use-case van de doelgrammatica.

Bij de aanvang van het project was als eerste doel gekozen: het kunnen parsen van minstens 200.000 regels Fortran code. Dit doel is ruimschoots bereikt maar als kwaliteitscriterium is het alleen zinvol als er ook iets met het resultaat kan worden gedaan. Dat kan alleen als de code zodanig door de parser ontleedt kan worden dat er voldoende structuur in is te herkennen. De Fortran standaard beschrijft deze structuur tot in detail en aangetoond is dat de parser deze structuur ook herkent. Bewerkingen (tranformaties) op Fortran code middels ASF vergelijkingen zijn mogelijk gebleken.

Het parsen van zoveel mogelijk bestaande Fortran code kan als test worden gebruikt. Het aantal regels testcode zegt echter niets over de diversiteit en daarmee de testkwaliteit van de testcode. Hiervoor is een coverage analyse toegepast. Aangetoond is dat met behulp van coverage-analyse testcode kan worden gevonden waarmee meer productieregels van de grammatica getest worden en ook meer fouten worden gevonden. Het coverage testresultaat van 44% geeft aanleiding tot meer vragen over de kwaliteit van de grammatica en ook over de taal Fortran. Fortran90 is een zeer omvangrijke taal waarvan veel taalelementen zelden gebruikt worden. Ook dat kan met een coverage test bepaald worden.

De Fortran90 grammatica is inmiddels een van de talen die in de sdf-library van de Meta-Environment worden meegeleverd. Gedurende het gebruik zullen er ongetwijfeld nog vele aanpassingen worden gedaan om het beter geschikt te maken als base-line grammar en om fouten en ommissies in de code te herstellen. De grammarware lifecycle van deze Fortran grammatica is begonnen.

Bibliografie

[1] Alfred V. Aho, R. S., and Ullman, J. D. *Compilers Principles, Techniques and Tools*. Prentice-Hall Inc., Upper Saddle River, N.J., USA, 1986.

[2] de Jong, H., and Olivier, P. Aterm library user manual.

[3] ISO. Iso/iec 1539:1991(e).

[4] Klint, P., Lämmel, R., and Verhoef, C. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology 14*, 3 (2005), 331–380.

[5] Lämmel, R. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001* (2001), vol. 2029 of *LNCS*, Springer-Verlag, pp. 201–216.

[6] Lämmel, R., and Verhoef, C. Semi-automatic Grammar Recovery. *Software—Practice & Experience 31*, 15 (December 2001), 1395–1438.

[7] Metcalf, M., Reid, J., and Cohen, M. *Fortran 95/2003 Explained*. Oxford University Press, Oxford, 2004.

[8] Metcalf, M., and Reid, J. K. *Fortran 90/95 explained (2nd ed.)*. Oxford University Press, Inc., New York, NY, USA, 1999.

[9] vd Brand, M., and Klint, P. ASF+SDF Meta-Environment User Manual.

Bijlage A

Voorbeeld uit de iso standaard

D.1.11 Program units

R1101 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1102 *program-stmt* **is** PROGRAM *program-name*

R1103 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

Constraint: In a *main-program*, the *execution-part* must not contain a RETURN statement or an ENTRY statement.

Constraint: The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, must be identical to the *program-name* specified in the *program-stmt*.

Constraint: An automatic object must not appear in the *specification-part* (R204) of a main program.

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1105 *module-stmt* **is** MODULE *module-name*

R1106 *end-module-stmt* **is** END [MODULE [*module-name*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the *module-name* specified in the *module-stmt*.

Constraint: A module *specification-part* must not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.

Constraint: An automatic object must not appear in the *specification-part* (R204) of a module.

R1107 *use-stmt* **is** USE *module-name* [, *rename-list*]
or USE *module-name* , ONLY : [*only-list*]

R1108 *rename* **is** *local-name* => *use-name*

R1109 *only* **is** *access-id*
or [*local-name* =>] *use-name*

Constraint: Each *access-id* must be a public entity in the module.

Constraint: Each *use-name* must be the name of a public entity in the module.

R1110 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R1111 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R1112 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the *block-data-stmt*.

Constraint: A *block-data specification-part* may contain only USE statements, type declaration statements, IMPLICIT statements, PARAMETER statements, derived-type definitions, and

Figuur A.1: One page of the Fortran language standard

Bijlage B

Coverage test: count productions in parsetree

De SGLR parser levert als resultaat een parsetree bestand waar ook de gebruikte grammatica-regels in zijn verwerkt. Onderstaande code leest dit bestand en produceert een frequentietabel van de gebruikte productieregels. De meestgebruikte productieregel is die voor `LAYOUT`. De parsetree van executeerbaar Fortran programma zal de `ProgramUnit` productie eenmalig bevatten.
De notatie is in de ATerm vorm, bijvoorbeeld:
`prod([cf(sort("MainProgram"))],cf(sort("ProgramUnit")),no-attrs) 1`
waarbij het laatste getal de frequentie aangeeft. In dit geval heeft het testprogramma een MainProgram, wat bij een verzameling subroutines niet het geval hoeft te zijn. Voor het programma is de ATerm User Manual gebruikt [2].

```
#include <stdio.h>
#include <aterm1.h>
#include <aterm2.h>
#include <MEPT.h>

ATermTable prodtable;
ATerm   one;

void addToTable(PT_Production at);
int count = 0;
```

```
void parseTerm(PT_Tree tree){
  PT_Production prod = PT_getTreeProd(tree);
  PT_Args children = PT_getTreeArgs(tree);
  PT_Tree head;

  /* ATwarning("%t\n", prod); */
  addToTable(prod);
  if(++count % 500 == 0) fprintf(stderr,"%d         \r",count);
  while (!PT_isArgsEmpty(children)) {
    head = PT_getArgsHead(children);
    if (PT_isTreeAmb(head)) {
      head = PT_getArgsHead(PT_getTreeArgs(head));
      ATwarning("AMBIGUITY! %t\n", PT_getTreeProd(head));
    }
    if (head && PT_isTreeAppl(head)) {
      parseTerm(head);
    }
    children = PT_getArgsTail(children);
  }
}

void addToTable(PT_Production at){
        ATermInt value;
        int ival;
        if((value = (ATermInt)ATtableGet(prodtable, at)) == NULL){
                ATtablePut(prodtable,at,(ATerm)ATmakeInt(1));
        }
        else {
                ival = ATgetInt(value) + 1;
                ATtablePut(prodtable,at,(ATerm)ATmakeInt(ival));
        }
}

void dumpTable(){
        ATermList keys = ATtableKeys(prodtable);
        ATermList tail = keys;
        ATerm head;
        ATermInt value;

        while(!ATisEmpty(tail)){
                head = ATgetFirst(tail);
                tail = ATgetNext(tail);
                ATprintf("%t ",head);
                value = (ATermInt)ATtableGet(prodtable, head);
                printf("%d\n",ATgetInt(value));
```

```
        }
}

int main(int argc, char *argv[])
{
        ATerm bottomOfStack; /* Used in initialisation of library */
        PT_ParseTree term;
        FILE *iofile;

        ATinit(argc, argv, &bottomOfStack); /* Initialise the ATerm library. */
        PT_initMEPTApi();

        prodtable = ATtableCreate(2000,75);
        one = (ATerm)ATmakeInt(1);

        if (!(iofile = fopen(argv[1], "rb"))) {
                ATwarning("Error reading file\n");
        }

        term = PT_ParseTreeFromTerm(ATreadFromFile(iofile));

        if (term == NULL) {
                ATerror("cannot read term file\n");
        }
        else {
                printf("Start treeparse\n");
                parseTerm(PT_getParseTreeTop(term));
                printf("Dumping table\n");
                dumpTable();
        }
        return 0; /* End of program. */
}
```

Bijlage C

Fortran 90 SDF grammatica

```
module languages/fortran/syntax/FortranLex
%%
%% Input restrictions:
%% -Make sure that the last statement at the end of the file has a '\n' to prevent a parse error.
%% -Userdefined operators (.XYZ.) are not handled yet
%% -include 'file' lines must first be processed or make it a comment (put a ! in front)
%%
%% F77 fixed format text can also be parsed if some pre-processing is done first,
%% Minimal changes needed for fixed form source handling:
%% a. change continuation symbols in 6th column by a & on the previous line (but not in a comment of course)
%%    in VIM regexp: :g/^     [^ ]/-1s/$/ \&/ followed by :%/^     [^ ]/      / (5 spaces+nonspace -> 6 spaces)
%% b. change every comment-symbol (* or C) in the first column into a !.
%%    in VIM regexp: %s/^[C\*]/!/


exports

sorts
  BinaryConstant Character Comment StartCommentBlock FixedComment
  Continuation Dop EOS HexConstant
  Icon Ident Label LblDef
  Letter OctalConstant
  Rcon ScalarIntLiteralConstant  Scon
  SconDouble SconSingle
%% Ident Aliasses:
  Name ArrayName ComponentName GenericName NamelistGroupName TypeName EndName CommonBlockName DummyArgName
  EntryName ExternalName FunctionName ImpliedDoVariable IntrinsicProcedureName ObjectName ProgramName
  SFDummyArgName SubroutineName SubroutineNameUse VariableName

lexical syntax

%% Everything following a ! is comment. With the \n it is an End of Statement
  "!" ~[\n]*   -> Comment
  (Comment? "\n" [\ \t]* )+  -> EOS
%% ; at end of line is allowed
  (";" [\ \t]*)+ Comment? "\n" [\ \t]*  -> EOS

%% Fixed source form comment rules (beware of code starting in 1st column!)
  ("\n" [Cc\*] ~[\n]*)+         -> FixedComment
   FixedComment "\n" [\ \t]*  -> EOS

%% commentlines at the start of a file/program are handled seperately in Fortran90.sdf
  EOS  -> StartCommentBlock

%% Statement seperator is also EOS (used seldomly)
  ([\ \t]* ";" [\ \t]*)+            -> EOS
%%
%% Continuation can optionally be seperated by comment. The & on the next line is also optional.
```

```
   "&" [\ \t]* ("!" ~[\n]*)? "\n" ( [\ \t]+ "&" )?  -> Continuation
   Continuation -> LAYOUT

%% Layout does NOT have a \n since that's part of EOS
   [\ \t] -> LAYOUT

%% number as label: max 5 digits with space in 6th pos
    [0-9][0-9][0-9][0-9][0-9]  -> Label
    [0-9][0-9][0-9][0-9]            -> Label
    [0-9][0-9][0-9]          -> Label
    [0-9][0-9]               -> Label
    [0-9]   -> Label


%%
  [A-Z]                         -> Letter
  [A-Za-z][A-Za-z0-9\_]*        -> Ident
%% Integer Constant
  [0-9]+                        -> Icon

%%R420 Character constants allow '' and "" as escaped ' and " in string literals like 'don''t' (="don't")
%% seperate names needed for lexical restrictions later
%% NOTE: continuated strings like 'hello&\n   &world' will be parsed as one string with & embedded.
  [\'] (~[\'] | "'")*  [\']    -> SconSingle
  [\"] (~[\"] | '""')*  [\"]    -> SconDouble
  SconSingle | SconDouble       -> Scon

%%R301: character definition with the 21 specials from table 3.1
  [a-zA-Z0-9\ \=\+\-\*\/\(\)\,\;\'\:\!\"\%\&\<\>\\\?\$] -> Character

%%R412-R416 Real and Double Literal Constant. Kind is handled in R400Datatypes.sdf
%% [sign] significant [exponent-letter exponent] [__kind-param]
%% JD: [\+\-]? removed because of Sign in R707
          [0-9]+ "." [0-9]* ([EeDd] [\+\-]? [0-9]+)?            -> Rcon
%% [sign] digit-string exponent-letter exponent [__kind-param]
                 "." [0-9]+ ([EeDd] [\+\-]? [0-9]+)?            -> Rcon
%% or [sign] digit-string exponent-letter exponent [__kind-param]
          [0-9]+ [EeDd] [\+\-]? [0-9]+                          -> Rcon
%%R408
  [Bb] [\'] [01]+ [\']          -> BinaryConstant
  [Bb] [\"] [01]+ [\"]          -> BinaryConstant
%%R409
  [Oo] [\'] [0-7]+ [\']         -> OctalConstant
  [Oo] [\"] [0-7]+ [\']         -> OctalConstant
%%R410
  [Zz] [\'] [0-9A-Fa-f]+ [\"]   -> HexConstant
  [Zz] [\"] [0-9A-Fa-f]+ [\"]   -> HexConstant

%%
  [0-9]+                        -> ScalarIntLiteralConstant

%%R704,724 Defined-unary-operator and Defined-binary-operator
%% JD: disabled Userdefined operators to prevent ambig with normal relops in expressions.
  "." "TODO" Letter+ "."        -> Dop

lexical restrictions
  SconSingle -/- [\']
  SconDouble -/- [\"]

%% commentline should be the complete line until newline. This must be lexical restriction
%%   Comment  -/- ~[\n]

%% if  (Comment "\n" [\ \t]* )+ -> EOS, then to prevent ambig with layout:
   EOS -/- [\ \t]

context-free restrictions

   Continuation? -/- ~[\n]

%% layout definition **without** \n due to the EOS definition. Lines are importent in Fortran.
%% Continuation-char & should be part of continuation, so not layout.
   LAYOUT? -/- [\ \t\&]

%% to prevent layout/comment parsed as after LblDef (which causes ambigs)
   LblDef -/- [\!]
```

```
context-free syntax
   Label?                               -> LblDef

%% Ident Aliasses. TODO: put them in the right modules.
  Ident  -> Name
  Ident  -> ArrayName
  Ident  -> ComponentName
  Ident  -> GenericName
  Ident  -> NamelistGroupName
  Ident  -> TypeName
  Ident  -> EndName
  Ident  -> CommonBlockName
  Ident  -> DummyArgName
  Ident  -> EntryName
  Ident  -> ExternalName
  Ident  -> FunctionName
  Ident  -> ImpliedDoVariable
  Ident  -> IntrinsicProcedureName
  Ident  -> ObjectName
  Ident  -> ProgramName
  Ident  -> SFDummyArgName
  Ident  -> SubroutineName
  Ident  -> SubroutineNameUse
  Ident  -> VariableName
%%%%%%%%%%
%% This Fortran grammar is structured according to document ISO/IEC 1539:1991.
%% Rulenumbers are named R2xx - R12xx as in Annex D of 1539:1991
%% The grammar is adapted from an ELI project grammer, http://eli-project.cvs.sourceforge.net/eli-project
%%
%%%%%%%%%%
%% ISO/IEC 1539:1991 section R2xx Fortran terms and concepts
%%%%%%%%%%

module languages/fortran/syntax/Fortran90

%%imports languages/fortran/syntax/FortranLex
imports languages/fortran/syntax/R400DataTypes
imports languages/fortran/syntax/R500DataDeclarations
imports languages/fortran/syntax/R600UseDataObjects
imports languages/fortran/syntax/R700Expressions
imports languages/fortran/syntax/R800ExecutionControl
imports languages/fortran/syntax/R900IOStatements
imports languages/fortran/syntax/R1000_IOEditing
imports languages/fortran/syntax/R1100ProgramUnits
imports languages/fortran/syntax/R1200Procedures

exports

sorts
  ActionStmt ArrayName Body
  BodyConstruct CommonBlockName ComponentName DeclarationConstruct DefinedOperator
  DummyArgName EndName EntryName ExecutableConstruct ExecutableProgram
  ExecutionPartConstruct ExternalName FunctionName GenericName ImpliedDoVariable
  InternalSubProgPart InternalSubprogram IntrinsicProcedureName  MainProgram
  MainRange  ModuleSubprogram ModuleSubprogramPartConstruct Name
  NamelistGroupName ObjectName ProgramName ProgramUnit SFDummyArgName
  SpecificationPartConstruct SpecificationStmt SubroutineName SubroutineNameUse TypeName
  VariableName


context-free start-symbols
  ExecutableProgram

context-free syntax

%%R201 JD: the optional startcommentblock is needed to parse comment lines in 1st lines of a file.
%% original: ExecutableProgram ProgramUnit -> ExecutableProgram
  StartCommentBlock? ProgramUnit+       -> ExecutableProgram

%%R202
%%R203
  MainProgram                           -> ProgramUnit
  FunctionSubprogram                    -> ProgramUnit
  SubroutineSubprogram                  -> ProgramUnit
  Module                                -> ProgramUnit
  BlockDataSubprogram                   -> ProgramUnit
```

```
%%R1101
  ProgramStmt? MainRange                -> MainProgram
  BodyConstruct+ EndProgramStmt         -> MainRange
  InternalSubProgPart EndProgramStmt    -> MainRange
  EndProgramStmt                        -> MainRange

  SpecificationPartConstruct            -> BodyConstruct
  ExecutableConstruct                   -> BodyConstruct
  BodyConstruct+                        -> Body

%%R204
  ImplicitStmt                          -> SpecificationPartConstruct
  ParameterStmt                         -> SpecificationPartConstruct
  FormatStmt                            -> SpecificationPartConstruct
  EntryStmt                             -> SpecificationPartConstruct
  DeclarationConstruct                  -> SpecificationPartConstruct
  UseStmt                               -> SpecificationPartConstruct

%%R207
  TypeDeclarationStmt                   -> DeclarationConstruct
  SpecificationStmt                     -> DeclarationConstruct
  DerivedTypeDef                        -> DeclarationConstruct
  InterfaceBlock                        -> DeclarationConstruct

%%R209
  ExecutableConstruct                   -> ExecutionPartConstruct
  FormatStmt                            -> ExecutionPartConstruct
  DataStmt                              -> ExecutionPartConstruct
  EntryStmt                             -> ExecutionPartConstruct

%%R214
  AccessStmt                            -> SpecificationStmt
  AllocatableStmt                       -> SpecificationStmt
  CommonStmt                            -> SpecificationStmt
  DataStmt                              -> SpecificationStmt
  DimensionStmt                         -> SpecificationStmt
  EquivalenceStmt                       -> SpecificationStmt
  ExternalStmt                          -> SpecificationStmt
  IntrinsicStmt                         -> SpecificationStmt
  SaveStmt                              -> SpecificationStmt
  IntentStmt                            -> SpecificationStmt
  NamelistStmt                          -> SpecificationStmt
  OptionalStmt                          -> SpecificationStmt
  PointerStmt                           -> SpecificationStmt
  TargetStmt                            -> SpecificationStmt

%%R210
  Body ContainsStmt   InternalSubprogram        -> InternalSubProgPart
  ContainsStmt        InternalSubprogram        -> InternalSubProgPart
  InternalSubProgPart InternalSubprogram        -> InternalSubProgPart

%%R211
  FunctionSubprogram            -> InternalSubprogram
  SubroutineSubprogram          -> InternalSubprogram

%%R212
  ContainsStmt                  -> ModuleSubprogramPartConstruct
  ModuleSubprogram              -> ModuleSubprogramPartConstruct

%%R213
  FunctionSubprogram            -> ModuleSubprogram
  SubroutineSubprogram          -> ModuleSubprogram

%%R215
  ActionStmt                    -> ExecutableConstruct
  DoConstruct                   -> ExecutableConstruct
  IfConstruct                   -> ExecutableConstruct
  CaseConstruct                 -> ExecutableConstruct
  WhereConstruct                -> ExecutableConstruct
  EndDoStmt                     -> ExecutableConstruct

%%R216
  AllocateStmt                  -> ActionStmt
  CycleStmt                     -> ActionStmt
```

```
  DeallocateStmt                -> ActionStmt
  ExitStmt                      -> ActionStmt
  NullifyStmt                   -> ActionStmt
  PointerAssignmentStmt         -> ActionStmt
  WhereStmt                     -> ActionStmt
  ArithmeticIfStmt              -> ActionStmt
  AssignmentStmt                -> ActionStmt
  AssignStmt                    -> ActionStmt
  BackspaceStmt                 -> ActionStmt
  CallStmt                      -> ActionStmt
  CloseStmt                     -> ActionStmt
  ContinueStmt                  -> ActionStmt
  EndfileStmt                   -> ActionStmt
  GotoStmt                      -> ActionStmt
  ComputedGotoStmt              -> ActionStmt
  AssignedGotoStmt              -> ActionStmt
  IfStmt                        -> ActionStmt
  InquireStmt                   -> ActionStmt
  OpenStmt                      -> ActionStmt
  PauseStmt                     -> ActionStmt
  PrintStmt                     -> ActionStmt
  ReadStmt                      -> ActionStmt
  ReturnStmt                    -> ActionStmt
  RewindStmt                    -> ActionStmt
%%  StmtFunctionStmt            -> ActionStmt see R1226 comment
  StopStmt                      -> ActionStmt
  WriteStmt                     -> ActionStmt


%%R311
  Dop                   -> DefinedOperator
  PowerOp               -> DefinedOperator
  MultOp                -> DefinedOperator
  AddOp                 -> DefinedOperator
  ConcatOp              -> DefinedOperator
  RelOp                 -> DefinedOperator
  NotOp                 -> DefinedOperator
  AndOp                 -> DefinedOperator
  OrOp                  -> DefinedOperator
  EquivOp               -> DefinedOperator
%%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 4xx DataTypes
%%%%%%%%%%

module languages/fortran/syntax/R400DataTypes

imports
  languages/fortran/syntax/FortranLex
  languages/fortran/syntax/R500DataDeclarations

exports

sorts

  AcImpliedDo AcValue AcValueList  ArrayConstructor
  BozLiteralConstant ComponentArraySpec ComponentAttrSpec ComponentAttrSpecList ComponentDecl
  ComponentDeclList ComponentDefStmt Constant DerivedTypeBody DerivedTypeDef
  DerivedTypeStmt EndTypeStmt KindParam LogicalConstant PrivateSequenceStmt
  StructureConstructor UnsignedArithmeticConstant

context-free syntax
%%-R401 lex: signed-digit-string is [sign] digit-string
%%-R402 lex: digit-string is digit [digit]...

%%R404
  Icon '_' KindParam                    -> UnsignedArithmeticConstant

%%R405
  Icon                                  -> KindParam
  NamedConstantUse                      -> KindParam

%%R406
  NamedConstantUse                      -> Constant
  UnsignedArithmeticConstant            -> Constant
  '+' UnsignedArithmeticConstant        -> Constant
  '-' UnsignedArithmeticConstant        -> Constant
```

```
  Scon                                -> Constant
%%  Hcon                                -> Constant
  LogicalConstant                     -> Constant

%%R420 char-literal-constant
  Icon '_' Scon                       -> Constant
  NamedConstantUse '_' Scon           -> Constant

%%R407
  BinaryConstant                      -> BozLiteralConstant
  OctalConstant                       -> BozLiteralConstant
  HexConstant                         -> BozLiteralConstant

%%R413
  Rcon '_' KindParam                  -> UnsignedArithmeticConstant

%%R421
  '.true.' '_' KindParam              -> LogicalConstant
  '.false.' '_' KindParam             -> LogicalConstant

%%R422
%% TODO: iso defines order: privateseq, componentdef.
  DerivedTypeStmt DerivedTypeBody+ EndTypeStmt          -> DerivedTypeDef
  PrivateSequenceStmt                                   -> DerivedTypeBody
  ComponentDefStmt                                      -> DerivedTypeBody

%%R423
  LblDef 'private' EOS                                  -> PrivateSequenceStmt
  LblDef 'sequence' EOS                                 -> PrivateSequenceStmt

%%R424
  LblDef 'type' TypeName EOS                            -> DerivedTypeStmt
  LblDef 'type' ':' ':' TypeName EOS                    -> DerivedTypeStmt
  LblDef 'type' ',' AccessSpec ':' ':' TypeName EOS     -> DerivedTypeStmt

%%R425
  LblDef 'end' 'type' TypeName? EOS                     -> EndTypeStmt

%%R426
  LblDef TypeSpec ( ',' ComponentAttrSpecList )? ':' ':' ComponentDeclList EOS          -> ComponentDefStmt
  LblDef TypeSpec ComponentDeclList EOS                                                 -> ComponentDefStmt

%%R427
  {ComponentAttrSpec ","}+                              -> ComponentAttrSpecList
  'pointer'                                             -> ComponentAttrSpec
  'dimension' '(' ComponentArraySpec ')'                -> ComponentAttrSpec

%%R428
  ExplicitShapeSpecList                                 -> ComponentArraySpec
  DeferredShapeSpecList                                 -> ComponentArraySpec

%%R429
%% TODO check with iso def
  ComponentName ('(' ComponentArraySpec ')')? ('*' CharLength)?         -> ComponentDecl
  {ComponentDecl ","}+                                                  -> ComponentDeclList

%%R430
 TypeName '(' {Expr ","}+ ')'                   -> StructureConstructor

%%R431
  '(/' AcValueList '/)'                         -> ArrayConstructor

%%R432
%%  Expr                                        -> AcValueList
%%  AcValueList1                                -> AcValueList
%%  Expr ',' Expr                               -> AcValueList1
%%  Expr ',' AcImpliedDo                        -> AcValueList1
%%  AcImpliedDo                                 -> AcValueList1
%%  AcValueList1 ',' Expr                       -> AcValueList1
%%  AcValueList1 ',' AcImpliedDo                -> AcValueList1
  Expr                                          -> AcValue
  AcImpliedDo                                   -> AcValue
  {AcValue ","}+                                -> AcValueList

%%R433
  '(' Expr ',' ImpliedDoVariable '=' Expr ',' Expr ')'                  -> AcImpliedDo
```

```
  '(' Expr ',' ImpliedDoVariable '=' Expr ',' Expr ',' Expr ')'         -> AcImpliedDo
  '(' AcImpliedDo ',' ImpliedDoVariable '=' Expr ',' Expr ')'           -> AcImpliedDo
  '(' AcImpliedDo ',' ImpliedDoVariable '=' Expr ',' Expr ',' Expr ')'  -> AcImpliedDo

%%%%%%%%%%
%%Fortran ISO/IEC 1539:1991 section R5xx Data Object declarations and Specifications
%%%%%%%%%%

module languages/fortran/syntax/R500DataDeclarations

imports
  languages/fortran/syntax/FortranLex
  languages/fortran/syntax/R400DataTypes
  languages/fortran/syntax/R700Expressions

exports

sorts
  AccessId AccessIdList AccessSpec AccessStmt AllocatableStmt
  ArrayAllocation ArrayAllocationList ArrayDeclarator ArrayDeclaratorList ArraySpec
  AssumedShapeSpec AssumedShapeSpecList AssumedSizeSpec AttrSpec CharLength
  CharSelector Comblock CommonBlockObject CommonBlockObjectList
  CommonStmt Constant DataIDoObject DataIDoObjectList DataImpliedDo
  Datalist DataStmt DataStmtObject DataStmtObjectList DataStmtSet
  DataStmtValue DataStmtValueList DeferredShapeSpec DeferredShapeSpecList DimensionStmt
  EntityDecl EquivalenceObject EquivalenceSet EquivalenceSetList EquivalenceStmt
  ExplicitShapeSpec ExplicitShapeSpecList ImplicitSpec ImplicitStmt IntentPar
  IntentParList IntentSpec IntentStmt KindSelector LengthSelector
  LetterSpec LowerBound NamedConstant NamedConstantDef NamedConstantDefList
  NamedConstantUse NamelistGroup NamelistGroupObject NamelistStmt
  OptionalPar OptionalParList OptionalStmt ParameterStmt PointerStmt
  PointerStmtObject PointerStmtObjectList SavedEntity SavedEntityList SaveStmt
  TargetObject TargetObjectList TargetStmt TypeDeclarationStmt TypeParamValue
  TypeSpec UpperBound

context-free syntax

%%R307
  Ident                         -> NamedConstant
  Ident                         -> NamedConstantUse

%%R501
  LblDef TypeSpec (',' AttrSpec)* ':' ':' {EntityDecl ","}+ EOS         -> TypeDeclarationStmt
  LblDef TypeSpec {EntityDecl ","}+ EOS                                 -> TypeDeclarationStmt

%%R502
  'integer'   KindSelector      -> TypeSpec
  'real'      KindSelector      -> TypeSpec
  'double' 'precision'          -> TypeSpec
  'complex'   KindSelector      -> TypeSpec
  'character' CharSelector      -> TypeSpec
  'logical'   KindSelector      -> TypeSpec
  'type' '(' TypeName ')'       -> TypeSpec
%%old F77 rules:
  'integer'                     -> TypeSpec
  'real'                        -> TypeSpec
  'complex'                     -> TypeSpec
  'logical'                     -> TypeSpec
  'character'                   -> TypeSpec
  'character' LengthSelector    -> TypeSpec

%%R503
  'parameter'                   -> AttrSpec
   AccessSpec                   -> AttrSpec
  'allocatable'                 -> AttrSpec
  'dimension' '(' ArraySpec ')' -> AttrSpec
  'external'                    -> AttrSpec
  'intent' '(' IntentSpec ')'   -> AttrSpec
  'intrinsic'                   -> AttrSpec
  'optional'                    -> AttrSpec
  'pointer'                     -> AttrSpec
  'save'                        -> AttrSpec
  'target'                      -> AttrSpec

%%R504
  ObjectName '=' Expr                                -> EntityDecl
```

```
  ObjectName '(' ArraySpec ')' '=' Expr                -> EntityDecl
  ObjectName '*' CharLength '=' Expr                   -> EntityDecl
  ObjectName '*' CharLength '(' ArraySpec ')' '=' Expr  -> EntityDecl
%%F77
  ObjectName                                           -> EntityDecl
  ObjectName '*' CharLength                            -> EntityDecl
  ObjectName '(' ArraySpec ')'                         -> EntityDecl
  ObjectName '(' ArraySpec ')' '*' CharLength          -> EntityDecl


%%R505
%% expr used as scalar-int-initialization-expr
  '(' ('kind' '=')? Expr ')'                           -> KindSelector
%%TODO: remove this often used in complex declarations like Integer*8 Complex*8 Complex*16 (IBM standard, not ISO!?)
  '*' Icon                                             -> KindSelector

%%R506
  '(' 'len' '=' TypeParamValue ',' 'kind' '=' Expr ')'  -> CharSelector
  '(' 'len' '=' TypeParamValue ',' Expr ')'             -> CharSelector
  '(' 'len' '=' TypeParamValue ')'                      -> CharSelector
  '(' ('kind' '=')? Expr ')'                            -> CharSelector

%%R507
  '(' TypeParamValue ')'        -> LengthSelector
%%F77
  '*' CharLength                -> LengthSelector

%%R508
  '(' TypeParamValue ')'        -> CharLength
  ScalarIntLiteralConstant      -> CharLength

%%R509
  SpecificationExpr | '*'       -> TypeParamValue

%%R510
  'public'      -> AccessSpec
  'private'     -> AccessSpec

%%R511
  'in'          -> IntentSpec
  'out'         -> IntentSpec
  'in' 'out'    -> IntentSpec

%%R512
  AssumedShapeSpecList                          -> ArraySpec
  DeferredShapeSpecList                         -> ArraySpec
  ExplicitShapeSpecList                         -> ArraySpec
  AssumedSizeSpec                               -> ArraySpec

%%R513
  {ExplicitShapeSpec ","}+                      -> ExplicitShapeSpecList
  (LowerBound ':')? UpperBound                  -> ExplicitShapeSpec

%%R514
  SpecificationExpr                             -> LowerBound

%%R515
  SpecificationExpr                             -> UpperBound

%%R516
  LowerBound? ':'                               -> AssumedShapeSpec
  LowerBound ':'                                -> AssumedShapeSpecList
  DeferredShapeSpecList ',' LowerBound ':'      -> AssumedShapeSpecList
  AssumedShapeSpecList ',' AssumedShapeSpec     -> AssumedShapeSpecList

%%R517
  {DeferredShapeSpec ","}+                      -> DeferredShapeSpecList
  ':'                                           -> DeferredShapeSpec

%%R518
  (LowerBound ':')? '*'                             -> AssumedSizeSpec
  ExplicitShapeSpecList ',' '*'                     -> AssumedSizeSpec
  ExplicitShapeSpecList ',' LowerBound ':' '*'      -> AssumedSizeSpec

%%R519
  LblDef 'intent' '(' IntentSpec ')' ':' ':' IntentParList EOS  -> IntentStmt
```

```
  LblDef 'intent' '(' IntentSpec ')' IntentParList EOS          -> IntentStmt
  {IntentPar ","}+                                              -> IntentParList
  DummyArgName                                                  -> IntentPar

%%R520
  LblDef 'optional' ':' ':' OptionalParList EOS         -> OptionalStmt
  LblDef 'optional' OptionalParList EOS                 -> OptionalStmt
  {OptionalPar ","}+                                    -> OptionalParList
  DummyArgName                                          -> OptionalPar

%%R521
  LblDef AccessSpec ':' ':' AccessIdList EOS            -> AccessStmt
  LblDef AccessSpec AccessIdList? EOS                   -> AccessStmt

%%R522
  {AccessId ","}+               -> AccessIdList
  GenericName                   -> AccessId
  GenericSpec                   -> AccessId

%%R523 see 26
  LblDef 'save' ':' ':' SavedEntityList EOS             -> SaveStmt
  LblDef 'save' SavedEntityList? EOS                    -> SaveStmt

%%R524
  {SavedEntity ","}+            -> SavedEntityList
  VariableName                  -> SavedEntity
  '/' CommonBlockName '/'       -> SavedEntity

%%R525
  LblDef 'dimension' ':' ':' ArrayDeclaratorList EOS    -> DimensionStmt
  LblDef 'dimension' ArrayDeclaratorList EOS            -> DimensionStmt

  {ArrayDeclarator ","}+                                -> ArrayDeclaratorList
  VariableName '(' ArraySpec ')'                        -> ArrayDeclarator

%%R526
  LblDef 'allocatable' ':' ':' ArrayAllocationList EOS  -> AllocatableStmt
  LblDef 'allocatable' ArrayAllocationList EOS          -> AllocatableStmt
  {ArrayAllocation ","}+                                -> ArrayAllocationList
  ArrayName ( '(' DeferredShapeSpecList ')' )?          -> ArrayAllocation

%%R527
  LblDef 'pointer' ':' ':' PointerStmtObjectList EOS    -> PointerStmt
  LblDef 'pointer' PointerStmtObjectList EOS            -> PointerStmt
  {PointerStmtObject ","}+                              -> PointerStmtObjectList
  ObjectName                                            -> PointerStmtObject
  ObjectName '(' DeferredShapeSpecList ')'              -> PointerStmtObject

%%R528
  LblDef 'target' ':' ':' TargetObjectList EOS          -> TargetStmt
  LblDef 'target' TargetObjectList EOS                  -> TargetStmt
  {TargetObject ","}+                                   -> TargetObjectList
  ObjectName                                            -> TargetObject
  ObjectName '(' ArraySpec ')'                          -> TargetObject

%%R529
  LblDef 'data' Datalist EOS                            -> DataStmt
  {DataStmtSet ","}+                                    -> Datalist

%%R530
  DataStmtObjectList '/' DataStmtValueList '/'          -> DataStmtSet

%%R531
  {DataStmtObject ","}+                                 -> DataStmtObjectList
  Variable                                              -> DataStmtObject
  DataImpliedDo                                         -> DataStmtObject

%%R532
  {DataStmtValue ","}+                                          -> DataStmtValueList
  Constant                                                      -> DataStmtValue
%%TODO  @<Scalar Integer Literal Constant@> '*' Constant        -> DataStmtValue
  NamedConstantUse '*' Constant                                 -> DataStmtValue

%%R533 see 100
  StructureConstructor                  -> Constant
  BozLiteralConstant                    -> Constant
```

```
%%R535
  '(' DataIDoObjectList ',' ImpliedDoVariable '=' Expr ',' Expr (',' Expr)? ')' -> DataImpliedDo

%%R536
  {DataIDoObject ","}+                -> DataIDoObjectList
  ArrayElement                        -> DataIDoObject
  DataImpliedDo                       -> DataIDoObject
  StructureComponent                  -> DataIDoObject

%%R538
  LblDef 'parameter' '(' NamedConstantDefList ')' EOS  -> ParameterStmt

%%R539
%% original iso: named-constant-def is named-constant = initialization-expr
  {NamedConstantDef ","}+             -> NamedConstantDefList
  NamedConstant '=' Expr              -> NamedConstantDef

%%R540
  LblDef 'implicit' 'none' EOS        -> ImplicitStmt
  LblDef 'implicit' {ImplicitSpec ","}+ EOS     -> ImplicitStmt

%%541
  TypeSpec "(" {LetterSpec ","}+ ")"  -> ImplicitSpec
  Letter ( "-" Letter )?              -> LetterSpec

%%R543
%% todo: Use SDF notation.
%%  LblDef 'namelist' NamelistGroups  EOS                          -> NamelistStmt
%%                    '/' NamelistGroupName '/' NamelistGroupObject   -> NamelistGroups
%%  NamelistGroups ','? '/' NamelistGroupName '/' NamelistGroupObject   -> NamelistGroups
%%  NamelistGroups ',' NamelistGroupObject                         -> NamelistGroups
  LblDef 'namelist' NamelistGroup ( ','? NamelistGroup)*  EOS    -> NamelistStmt
  '/' NamelistGroupName '/' {NamelistGroupObject ","}+           -> NamelistGroup

%%R544
  VariableName                                          -> NamelistGroupObject

%%R545
  LblDef 'equivalence' EquivalenceSetList EOS           -> EquivalenceStmt

%%R546
  {EquivalenceSet ","}+                                 -> EquivalenceSetList
  '(' EquivalenceObject "," {EquivalenceObject ","}+ ')'        -> EquivalenceSet

%%R547 Note: Variable includes Substring variables %% TODO: check EqObject def substring,arrayname?
  ArrayName                             -> EquivalenceObject
  Variable                              -> EquivalenceObject

%%R548
%%  LblDef 'common' Comlist EOS         -> CommonStmt
%%  CommonBlockObject                   -> Comlist
%%  Comblock CommonBlockObject          -> Comlist
%%  Comlist ',' CommonBlockObject       -> Comlist
%%  Comlist Comblock CommonBlockObject  -> Comlist
%%  Comlist ',' Comblock CommonBlockObject      -> Comlist
%%  '/' '/'                             -> Comblock
%%  '/' CommonBlockName '/'             -> Comblock
  LblDef 'common' Comblock? CommonBlockObjectList (','? Comblock CommonBlockObjectList)* EOS    -> CommonStmt
  '/' CommonBlockName? '/'                                                                      -> Comblock

%%R549
%%todo: check if this can be reverted to
%%      original iso def: CommonBlockObject is VariableName [(ExplicitShapeSpecList)]
  VariableName                          -> CommonBlockObject
  ArrayDeclarator                       -> CommonBlockObject
  {CommonBlockObject ","}+              -> CommonBlockObjectList

%%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 section R6xx Use of Data Objects
%%%%%%%%%%

module languages/fortran/syntax/R600UseDataObjects

imports
  languages/fortran/syntax/FortranLex
```

```
  languages/fortran/syntax/R700Expressions

exports

sorts
  AllocatedShape AllocateObject AllocateObjectList AllocateShapeSpec AllocateStmt
  Allocation AllocationList ArrayElement DataRef DeallocateStmt
  FieldSelector NullifyStmt PointerField PointerObject PointerObjectList
  ScalarVariable SectionSubscript SectionSubscriptList StructureComponent Subscript
  SubscriptTriplet SubstringRange Variable

context-free syntax

%%R601
  VariableName | ArrayElement                                    -> ScalarVariable
  VariableName                                                    -> Variable
  VariableName '(' {Subscript ","}+ ')'                           -> Variable
  VariableName SubstringRange                                     -> Variable
  VariableName '(' {Subscript ","}+ ')' SubstringRange            -> Variable
  Expr                                                            -> Subscript

%%R612
  Name '%' Name                                                   -> DataRef
  DataRef '%' Name                                                -> DataRef
  Name '(' SectionSubscriptList ')'                               -> DataRef
  DataRef '(' SectionSubscriptList ')'                            -> DataRef

  {SectionSubscript ","}+                                         -> SectionSubscriptList
  Expr                                                            -> SectionSubscript
  SubscriptTriplet                                                -> SectionSubscript

%%R609
%%R610
%%R611
  '(' SubscriptTriplet ')'                                        -> SubstringRange

%%R614
  VariableName FieldSelector                                      -> StructureComponent
  StructureComponent FieldSelector                                -> StructureComponent
  '(' SectionSubscriptList ')' '%' Name                           -> FieldSelector
  '%' Name                                                        -> FieldSelector

%%R615
  StructureComponent '(' SectionSubscriptList ')'                 -> ArrayElement
  VariableName '(' SectionSubscriptList ')'                       -> ArrayElement

%%R619
  Expr? ':' Expr? (':' Expr)?                                     -> SubscriptTriplet

%%R622
  LblDef 'allocate' '(' AllocationList ',' 'stat' '=' Variable ')' EOS  -> AllocateStmt
  LblDef 'allocate' '(' AllocationList ')' EOS                          -> AllocateStmt

%%R623 chain rule deleted

%%R624
   {Allocation ","}+                          -> AllocationList
   AllocateObject AllocatedShape?             -> Allocation
   '(' SectionSubscriptList ')'               -> AllocatedShape
%%/* From ELI: Need to use SectionSubscriptList here to solve an LALR(1) conflict with the
%% * FieldSelector in R625.  (Can't tell which we have until the character
%% * following the right paren, but we must reduce WITHIN the parens.)
%% */

%%R625
  {AllocateObject ","}+                      -> AllocateObjectList
  VariableName                               -> AllocateObject
  AllocateObject FieldSelector               -> AllocateObject

%%R626
%% JD is this correct??/* Omitted to solve LALR(1) conflict.  see R624
  {Expr ":" }+                               -> AllocateShapeSpec

%%R629
  LblDef 'nullify' '(' PointerObjectList ')' EOS        -> NullifyStmt
  {PointerObject ","}+                                  -> PointerObjectList
```

```
%%R630
  Name                                      -> PointerObject
  PointerField                              -> PointerObject
  Name '(' SFExprList ')' '%' Name          -> PointerField
  Name '(' SFDummyArgNameList ')' '%' Name  -> PointerField
  Name '%' Name                             -> PointerField
  PointerField FieldSelector                -> PointerField

%%R631
  LblDef 'deallocate' '(' AllocateObjectList ',' 'stat' '=' Variable ')' EOS   -> DeallocateStmt
  LblDef 'deallocate' '(' AllocateObjectList ')' EOS                           -> DeallocateStmt


%%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 section R7xx Expressions and Assignments
%%%%%%%%%%

module languages/fortran/syntax/R700Expressions

imports
  languages/fortran/syntax/FortranLex
  languages/fortran/syntax/R1200Procedures
  languages/fortran/syntax/R600UseDataObjects

exports

sorts
  AddOp AddOperand AndOp AndOperand AssignmentStmt
  CExpr ComplexConst ConcatOp COperand CPrimary
  DefinedBinaryOp DefinedUnaryOp ElsewhereStmt EndWhereStmt
  EquivOp EquivOperand Expr Level1Expr Level2Expr
  Level3Expr Level4Expr Level5Expr LogicalConstant MaskExpr
  MultOp MultOperand NotOp  OrOp OrOperand
  PointerAssignmentStmt PowerOp Primary RelOp SFExpr
  SFExprList SFFactor SFPrimary SFTerm Sign
  SpecificationExpr Target UFExpr UFFactor UFPrimary
  UFTerm UnsignedArithmeticConstant WhereConstruct WhereConstructStmt
  WhereStmt

context-free syntax
%%%% TODO: put these under the right section

  UFTerm                        -> UFExpr
  Sign UFTerm                   -> UFExpr
  UFExpr AddOp UFTerm           -> UFExpr
  UFFactor                      -> UFTerm
  UFTerm MultOp UFFactor        -> UFTerm
  UFTerm ConcatOp UFPrimary     -> UFTerm
  UFPrimary                     -> UFFactor
  UFPrimary PowerOp UFFactor    -> UFFactor

  Icon                          -> UFPrimary
  Scon                          -> UFPrimary
  Name                          -> UFPrimary
  FunctionReference             -> UFPrimary
  DataRef                       -> UFPrimary
  '(' UFExpr ')'                -> UFPrimary

%%82,83
  (CExpr ConcatOp)? CPrimary    -> CExpr
  COperand                      -> CPrimary
  '(' CExpr ')'                 -> CPrimary
  Scon                          -> COperand
  Name                          -> COperand
  DataRef                       -> COperand
  FunctionReference             -> COperand

%%101
  Icon                          -> UnsignedArithmeticConstant
  Rcon                          -> UnsignedArithmeticConstant
  ComplexConst                  -> UnsignedArithmeticConstant

%%107
  '(' Expr ',' Expr ')'         -> ComplexConst
```

```
%%108
  '.true.' | '.false.'          -> LogicalConstant

%%R701
  ArrayConstructor              -> Primary
  ArrayConstructor              -> SFPrimary
  UnsignedArithmeticConstant    -> Primary
  Name                          -> Primary
  DataRef                       -> Primary
  FunctionReference             -> Primary
  '(' Expr ')'                  -> Primary
  Scon                          -> Primary

%%R703
  DefinedUnaryOp? Primary       -> Level1Expr

%%R704
  Dop                           -> DefinedUnaryOp

%%R705
  Level1Expr (PowerOp MultOperand)?     -> MultOperand

%%R706
  (AddOperand MultOp)? MultOperand      -> AddOperand

%%R707 % ELI: "We need to distinguish unary operators" SDF: ambig on "-2" on next 2 lines
  (Level2Expr AddOp)? AddOperand        -> Level2Expr
  Sign AddOperand                       -> Level2Expr

%%R708
  '**'                                  -> PowerOp

%%R709
  '*' | '/'                             -> MultOp

%%R710
  '+' | '-'                             -> AddOp
  '+' | '-'                             -> Sign

%%R711
  (Level3Expr ConcatOp)? Level2Expr     -> Level3Expr

%%R712
  '//'                                  -> ConcatOp

%%R713
  (Level3Expr RelOp)? Level3Expr        -> Level4Expr

%%R714
  '==' | '/=' | '<' | '<=' | '>' | '>='                         -> RelOp
%%from F77
  '.eq.' | '.ne.' | '.lt.' | '.le.' | '.gt.' | '.ge.'          -> RelOp

%%R715
  NotOp? Level4Expr                     -> AndOperand

%%R716
  (OrOperand AndOp)? AndOperand         -> OrOperand

%%R717
  (EquivOperand OrOp)? OrOperand        -> EquivOperand

%%R718
  (Level5Expr EquivOp)? EquivOperand    -> Level5Expr

%%R719
 '.not.'                                -> NotOp

%%R720
  '.and.'                               -> AndOp

%%R721
  '.or.'                                -> OrOp

%%R722
  '.eqv.' | '.neqv.'                    -> EquivOp
```

```
  LogicalConstant                       -> Primary

%%R723
  (Expr DefinedBinaryOp)? Level5Expr    -> Expr

%%R724
  Dop                                   -> DefinedBinaryOp

%%R725-R734 chain rule deleted

%%R734 %% JD seems nice to see difference EXPR and SpecificationEpr?
  Expr -> SpecificationExpr

%%R735
%% todo: check. Original ISO: assignmentStatement is variable '=' expression
  LblDef Name '%' Name '=' Expr EOS                                     -> AssignmentStmt
  LblDef Name '%' DataRef '=' Expr EOS                                  -> AssignmentStmt
  LblDef Name '(' SFExprList ')' '%' Name '=' Expr EOS                  -> AssignmentStmt
  LblDef Name '(' SFExprList ')' '%' DataRef '=' Expr EOS               -> AssignmentStmt
  LblDef Name '(' SFDummyArgNameList ')' '%' Name '=' Expr EOS          -> AssignmentStmt
  LblDef Name '(' SFDummyArgNameList ')' '%' DataRef '=' Expr EOS       -> AssignmentStmt
  LblDef Name '=' Expr EOS                                              -> AssignmentStmt
  LblDef Name '(' SFExprList ')' '=' Expr EOS                           -> AssignmentStmt
  LblDef Name '(' SFExprList ')' SubstringRange '=' Expr EOS            -> AssignmentStmt

  SFExpr ':' Expr ':' Expr                    -> SFExprList
  SFExpr ':' ':' Expr                         -> SFExprList
  ':' Expr ':' Expr                           -> SFExprList
  ':' ':' Expr                                -> SFExprList
  ':'                                         -> SFExprList
  ':' Expr                                    -> SFExprList
  SFExpr                                      -> SFExprList
  SFExpr ':'                                  -> SFExprList
  SFExpr ':' Expr                             -> SFExprList
  SFExprList ',' SectionSubscript             -> SFExprList
  SFDummyArgNameList ',' ':'                  -> SFExprList
  SFDummyArgNameList ',' ':' Expr             -> SFExprList
%% problem with code like P(A,2). A can be both SFExprList and SFDummyargnameList.
%% Conflict with 3 lines above: SFExprList ',' SectionSubscript
%%  SFDummyArgNameList ',' SFExpr               -> SFExprList
  SFDummyArgNameList ',' SFExpr ':'           -> SFExprList
  SFDummyArgNameList ',' SFExpr ':' Expr      -> SFExprList

  SFTerm                                      -> SFExpr
  Sign AddOperand                             -> SFExpr
  SFExpr AddOp AddOperand                     -> SFExpr
  SFFactor                                    -> SFTerm
  SFTerm MultOp MultOperand                   -> SFTerm
  SFPrimary                                   -> SFFactor
  SFPrimary PowerOp MultOperand               -> SFFactor
  Icon                                        -> SFPrimary
  Name                                        -> SFPrimary
  DataRef                                     -> SFPrimary
  FunctionReference                           -> SFPrimary
  '(' Expr ')'                                -> SFPrimary

%%R736
  LblDef Name '=>' Target EOS                                           -> PointerAssignmentStmt
  LblDef Name '%' Name '=>' Target EOS                                  -> PointerAssignmentStmt
  LblDef Name '%' DataRef '=>' Target EOS                               -> PointerAssignmentStmt
  LblDef Name '(' SFExprList ')' '%' Name '=>' Target EOS               -> PointerAssignmentStmt
  LblDef Name '(' SFExprList ')' '%' DataRef '=>' Target EOS            -> PointerAssignmentStmt
  LblDef Name '(' SFDummyArgNameList ')' '%' Name '=>' Target EOS       -> PointerAssignmentStmt
  LblDef Name '(' SFDummyArgNameList ')' '%' DataRef '=>' Target EOS    -> PointerAssignmentStmt

%%R737
  Expr  -> Target

%%R738
  LblDef 'where' '(' MaskExpr ')' AssignmentStmt  -> WhereStmt

%%R739
%% todo: sdf-fy
%%  Where     EndWhereStmt      -> WhereConstruct
%%  ElseWhere EndWhereStmt      -> WhereConstruct
```

```
%%  WhereConstructStmt          -> Where
%%  Where AssignmentStmt        -> Where
%%  Where ElsewhereStmt         -> ElseWhere
%%  ElseWhere AssignmentStmt    -> ElseWhere
  WhereConstructStmt AssignmentStmt*
              (ElsewhereStmt AssignmentStmt*)?
       EndWhereStmt                          -> WhereConstruct

%%R740
  LblDef 'where' '(' MaskExpr ')' EOS        -> WhereConstructStmt

%%R741
  Expr  -> MaskExpr

%%R742
  LblDef 'elsewhere' EOS                     -> ElsewhereStmt

%%R743
  LblDef 'end' 'where' EOS                   -> EndWhereStmt

%%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 section R8xx Execution Control
%%%%%%%%%%

module languages/fortran/syntax/R800ExecutionControl

imports
  languages/fortran/syntax/FortranLex
  languages/fortran/syntax/Fortran90

exports

sorts
  ArithmeticIfStmt AssignedGotoStmt AssignStmt BlockDoConstruct CaseBodyConstruct
  CaseConstruct CaseSelector CaseStmt CaseValueRange ComputedGotoStmt
  ContinueStmt CycleStmt DoConstruct ElseIfStmt ElseStmt
  EndDoStmt EndIfStmt EndSelectStmt ExitStmt GoToKw
  GotoStmt IfConstruct  IfConstructName IfStmt IfThenStmt
  Int-Real-Dp-Expression LabelDoStmt  LblRef LoopControl PauseStmt
  ScalarIntExpr ScalarLogicalExpr ScalarNumericExpr SelectCaseBody SelectCaseRange
  StopStmt

context-free syntax

%%R802
  IfThenStmt ExecutionPartConstruct*
              (ElseIfStmt ExecutionPartConstruct*)*
              (ElseStmt   ExecutionPartConstruct*)?
        EndIfStmt                                                  -> IfConstruct

%%R803
  LblDef (IfConstructName ":")? 'if' '(' ScalarLogicalExpr ')' 'then' EOS      -> IfThenStmt
  Ident -> IfConstructName

%%R804
  LblDef 'else' 'if' '(' ScalarLogicalExpr ')' 'then' IfConstructName? EOS     -> ElseIfStmt

%%R805
  LblDef 'else' IfConstructName? EOS                                           -> ElseStmt

%%R806
  LblDef 'end' 'if' IfConstructName? EOS                                       -> EndIfStmt

%%R807
%% JD: removed EOS at end since its part of ActionStmt
  LblDef 'if' '(' ScalarLogicalExpr ')' ActionStmt                             -> IfStmt
%% JD: simplification
  Expr -> ScalarLogicalExpr

%%R808
%%  LblDef Name ':' 'selectcase' '(' Expr ')' EOS SelectCaseRange      -> CaseConstruct
%%  LblDef 'selectcase' '(' Expr ')' EOS SelectCaseRange               -> CaseConstruct
  LblDef Name ':' 'select' 'case' '(' Expr ')' EOS SelectCaseRange     -> CaseConstruct
  LblDef 'select' 'case' '(' Expr ')' EOS SelectCaseRange              -> CaseConstruct

  SelectCaseBody EndSelectStmt                                -> SelectCaseRange
```

```
  EndSelectStmt                                                -> SelectCaseRange

  CaseBodyConstruct+                            -> SelectCaseBody
  CaseStmt                                      -> CaseBodyConstruct
  ExecutionPartConstruct                        -> CaseBodyConstruct

%%R810
  LblDef 'case' CaseSelector Name? EOS          -> CaseStmt

%%R811
  LblDef 'end' 'select' EndName? EOS            -> EndSelectStmt

%%R813
  '(' { CaseValueRange "," }+  ')'              -> CaseSelector
  'default'                                     -> CaseSelector

%%R814
  Expr                                          -> CaseValueRange
  Expr ':'                                      -> CaseValueRange
  ':' Expr                                      -> CaseValueRange
  Expr ':' Expr                                 -> CaseValueRange

%%R816
   BlockDoConstruct                             -> DoConstruct

%%R817
%%/* Block DO constructs cannot be recognized syntactically because there is
%% * no requirement that there is an end do statement. (A do loop may use label+continue construct)
%%   DoStmt Block EndDoStmt  -> BlockDoConstruct
%%   DoStmt Block  -> BlockDoConstruct
%% JD: endo IS compulsory in cases where LblRef is missing. Can we use this to locate Do-blocks?

%%R818
  LblDef 'do' LblRef EOS                                -> BlockDoConstruct
  LblDef 'do' LoopControl EOS                           -> BlockDoConstruct
  LblDef 'do' EOS                                       -> BlockDoConstruct
  LblDef Name ':' 'do' LblRef LoopControl EOS           -> BlockDoConstruct
  LblDef Name ':' 'do' LblRef EOS                       -> BlockDoConstruct
  LblDef Name ':' 'do' LoopControl EOS                  -> BlockDoConstruct
  LblDef Name ':' 'do' EOS                              -> BlockDoConstruct

%%40
%%R818
  LabelDoStmt                                           -> DoConstruct

%%R819
  LblDef 'do' LblRef ','? LoopControl EOS               -> LabelDoStmt

%%R821
  'while' '(' Expr ')'                                                                                  -> LoopControl
  VariableName '=' Int-Real-Dp-Expression ',' Int-Real-Dp-Expression (',' Int-Real-Dp-Expression)?      -> LoopControl
%% TODO: Constraint: Int-Real-Dp-Expression should be a int, default-real or double-precision expression
  Expr                                                  -> Int-Real-Dp-Expression

%%R822
%% do-variable is scalar-variable

%%R825
  LblDef 'end' 'do' Name? EOS                           -> EndDoStmt

%%R834
  LblDef 'cycle' EndName? EOS                           -> CycleStmt

%%R835
  LblDef 'exit' EndName? EOS                            -> ExitStmt

%%R836
 'go' 'to'                                              -> GoToKw
  LblDef GoToKw LblRef EOS                              -> GotoStmt

%%R837
  LblDef GoToKw '(' {LblRef ","}+ ')' ","? ScalarIntExpr EOS  -> ComputedGotoStmt
  Icon  ->  LblRef
  Expr   -> ScalarIntExpr

%%R838
```

```
  LblDef 'assign' LblRef 'to' VariableName EOS  -> AssignStmt

%%R839
  LblDef GoToKw VariableName EOS                                -> AssignedGotoStmt
  LblDef GoToKw VariableName ','? '(' {LblRef ","}+ ')' EOS     -> AssignedGotoStmt

%%R840
  LblDef 'if' '(' ScalarNumericExpr ')' LblRef ',' LblRef ',' LblRef EOS  -> ArithmeticIfStmt
  Expr -> ScalarNumericExpr

%%R841
  LblDef 'continue' EOS                        -> ContinueStmt

%%R842
  LblDef 'stop' (Icon | Scon)? EOS             -> StopStmt

%%R844
  LblDef 'pause' (Icon | Scon)? EOS            -> PauseStmt

%%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 section R9xx Input/Output Statements
%%%%%%%%%%

module languages/fortran/syntax/R900IOStatements

imports
  languages/fortran/syntax/FortranLex
  languages/fortran/syntax/Fortran90

exports

sorts
  BackspaceStmt CloseSpec CloseSpecList CloseStmt ConnectSpec
  ConnectSpecList EndfileStmt FormatIdentifier InputImpliedDo InputItem
  InputItemList InquireSpec InquireSpecList InquireStmt IoControlSpec
  IoControlSpecList OpenStmt OutputImpliedDo OutputItem OutputItemList
  PositionSpec PrintStmt RdCtlSpec RdFmtId RdFmtIdExpr
  RdIoCtlSpecList RdUnitId ReadStmt RewindStmt UnitIdentifier
  WriteStmt

context-free syntax
  %% already defined in R800 section:
  Icon  ->  LblRef

%%R901
  UFExpr | '*'                 -> UnitIdentifier

%%-R902 rule deleted:
%% external-file-unit is scalar-int-expr
%%-R903 rule deleted:
%% internal-file-unit is default-char-variable

%%R904
  LblDef 'open' '(' ConnectSpecList ')' EOS  -> OpenStmt

%%R905
  {ConnectSpec ","}+                  -> ConnectSpecList
  UnitIdentifier                      -> ConnectSpec
  'unit'     '=' UnitIdentifier       -> ConnectSpec
  'iostat'   '=' ScalarVariable       -> ConnectSpec
  'err'      '=' LblRef               -> ConnectSpec
  'file'     '=' CExpr                -> ConnectSpec
  'status'   '=' CExpr                -> ConnectSpec
  'access'   '=' CExpr                -> ConnectSpec
  'form'     '=' CExpr                -> ConnectSpec
  'recl'     '=' Expr                 -> ConnectSpec
  'blank'    '=' CExpr                -> ConnectSpec
%%since F90:
  'position' '=' CExpr                -> ConnectSpec
  'action'   '=' CExpr                -> ConnectSpec
  'delim'    '=' CExpr                -> ConnectSpec
  'pad'      '=' CExpr                -> ConnectSpec


%%-R906 rule deleted:
%% file-name-expr is scalar-default-char-variable
```

```
%%R907
  LblDef 'close' '(' CloseSpecList ')' EOS      -> CloseStmt

%%R908
  {CloseSpec ","}+                              -> CloseSpecList
              UnitIdentifier                    -> CloseSpec
  'unit'   '=' UnitIdentifier                   -> CloseSpec
  'iostat' '=' ScalarVariable                   -> CloseSpec
  'err'    '=' LblRef                           -> CloseSpec
  'status' '=' CExpr                            -> CloseSpec

%%909
  LblDef 'read' RdCtlSpec InputItemList? EOS    -> ReadStmt
  LblDef 'read' RdFmtId EOS                     -> ReadStmt
  LblDef 'read' RdFmtId ',' InputItemList EOS   -> ReadStmt

%%R910
  LblDef 'write' '(' IoControlSpecList ')' OutputItemList? EOS  -> WriteStmt

%%R911
  LblDef 'print' FormatIdentifier ( ',' OutputItemList )? EOS   -> PrintStmt

%%R912
  'unit'   '=' UnitIdentifier           -> IoControlSpec
  'fmt'    '=' FormatIdentifier         -> IoControlSpec
  'nml'    '=' NamelistGroupName        -> IoControlSpec
  'rec'    '=' Expr                     -> IoControlSpec
  'iostat' '=' ScalarVariable           -> IoControlSpec
  'err'    '=' LblRef                   -> IoControlSpec
  'end'    '=' LblRef                   -> IoControlSpec
  'advance' '='  CExpr                  -> IoControlSpec
  'size'   '=' Variable                 -> IoControlSpec
  'eor'    '=' LblRef                   -> IoControlSpec

  UnitIdentifier ',' FormatIdentifier?  -> IoControlSpecList
  UnitIdentifier ',' IoControlSpec      -> IoControlSpecList
  IoControlSpec                         -> IoControlSpecList
  IoControlSpecList ',' IoControlSpec   -> IoControlSpecList

%%912
  RdUnitId                              -> RdCtlSpec
  '(' RdIoCtlSpecList ')'               -> RdCtlSpec
  '(' UFExpr ')'                        -> RdUnitId
  '(' '*' ')'                           -> RdUnitId

  UnitIdentifier ',' IoControlSpec      -> RdIoCtlSpecList
  UnitIdentifier ',' FormatIdentifier   -> RdIoCtlSpecList
  IoControlSpec                         -> RdIoCtlSpecList
  RdIoCtlSpecList ',' IoControlSpec     -> RdIoCtlSpecList

%%R913 format
  LblRef                                -> RdFmtId
  '*'                                   -> RdFmtId
  COperand                              -> RdFmtId
  COperand ConcatOp CPrimary            -> RdFmtId
  RdFmtIdExpr ConcatOp CPrimary         -> RdFmtId
  '(' UFExpr ')'                        -> RdFmtIdExpr

%%913
  LblRef | CExpr | '*'                  -> FormatIdentifier

%%R914
  Name                                  -> InputItem
  DataRef                               -> InputItem
  InputImpliedDo                        -> InputItem
  {InputItem ","}+                      -> InputItemList

%%R915
  Expr                                  -> OutputItem
  OutputImpliedDo                       -> OutputItem
  {OutputItem ","}+                     -> OutputItemList

%%R916
%%R917
%%R918
```

```
  '(' InputItemList  ',' ImpliedDoVariable '=' Expr ',' Expr ')'               -> InputImpliedDo
  '(' InputItemList  ',' ImpliedDoVariable '=' Expr ',' Expr ',' Expr ')'      -> InputImpliedDo

  '(' OutputItemList ',' ImpliedDoVariable '=' Expr ',' Expr ')'               -> OutputImpliedDo
  '(' OutputItemList ',' ImpliedDoVariable '=' Expr ',' Expr ',' Expr ')'      -> OutputImpliedDo

%%R919
  LblDef 'backspace' UnitIdentifier EOS                 -> BackspaceStmt
  LblDef 'backspace' '(' {PositionSpec ","}+ ')' EOS    -> BackspaceStmt

%%R920
%% note: 'endfile' without space is allowed
  LblDef 'end' 'file' UnitIdentifier EOS                        -> EndfileStmt
  LblDef 'end' 'file' '('  {PositionSpec ","}+ ')' EOS          -> EndfileStmt

%%R921
  LblDef 'rewind' UnitIdentifier EOS                    -> RewindStmt
  LblDef 'rewind' '(' {PositionSpec ","}+ ')' EOS       -> RewindStmt

%%R922
  'unit='?  UnitIdentifier  -> PositionSpec
  'iostat=' ScalarVariable  -> PositionSpec
  'err='    LblRef          -> PositionSpec

%%R923
  LblDef 'inquire' '(' InquireSpecList ')' EOS                                  -> InquireStmt
  LblDef 'inquire' '(' 'iolength' '=' ScalarVariable ')' OutputItemList EOS     -> InquireStmt

%%R924
  'unit'        '=' UnitIdentifier      -> InquireSpec
  'file'        '=' CExpr               -> InquireSpec
  'iostat'      '=' ScalarVariable      -> InquireSpec
  'err'         '=' LblRef              -> InquireSpec
  'exist'       '=' ScalarVariable      -> InquireSpec
  'opened'      '=' ScalarVariable      -> InquireSpec
  'number'      '=' ScalarVariable      -> InquireSpec
  'named'       '=' ScalarVariable      -> InquireSpec
  'name'        '=' ScalarVariable      -> InquireSpec
  'access'      '=' ScalarVariable      -> InquireSpec
  'sequential'  '=' ScalarVariable      -> InquireSpec
  'direct'      '=' ScalarVariable      -> InquireSpec
  'form'        '=' ScalarVariable      -> InquireSpec
  'formatted'   '=' ScalarVariable      -> InquireSpec
  'unformatted' '=' ScalarVariable      -> InquireSpec
  'recl'        '=' Expr                -> InquireSpec
  'nextrec'     '=' ScalarVariable      -> InquireSpec
  'blank'       '=' ScalarVariable      -> InquireSpec
  'position'    '=' ScalarVariable      -> InquireSpec
  'action'      '=' ScalarVariable      -> InquireSpec
  'read'        '=' ScalarVariable      -> InquireSpec
  'write'       '=' ScalarVariable      -> InquireSpec
  'readwrite'   '=' ScalarVariable      -> InquireSpec
  'delim'       '=' ScalarVariable      -> InquireSpec
  'pad'         '=' ScalarVariable      -> InquireSpec

%%R924
%% see constraint: the unit= may be ommitted from the first spec in the list
  UnitIdentifier "," {InquireSpec ","}+ -> InquireSpecList
  {InquireSpec ","}+                    -> InquireSpecList

%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 section R10xx Input/Output Editing
%%%%%%%%%

module languages/fortran/syntax/R1000_IOEditing

imports
  %% needed for Icon, Scon, LblDef:
  languages/fortran/syntax/FortranLex

exports

sorts
  BlankInterpEditDescr CharStringEditDescr CharStringEditDescr  ControlEditDescr ControlEditDescr
  DataEditDescr DataEditDescr   FormatItem FormatItemList FormatStmt
  PositionEditDescr SignEditDescr
```

```
context-free syntax

%%R1001
%%R1002
  LblDef 'format' '(' FormatItemList? ')' EOS  -> FormatStmt

  { FormatItem ","}+                    -> FormatItemList

%%R1003
  Icon? DataEditDescr                   -> FormatItem
  ControlEditDescr                      -> FormatItem
  CharStringEditDescr                   -> FormatItem
  Icon? DataEditDescr                   -> FormatItem
  Icon? '(' FormatItemList ')'          -> FormatItem

%%R1005-R1009
  'I'  Icon ('.' Icon)?                 -> DataEditDescr
  'O'  Icon ('.' Icon)?                 -> DataEditDescr
  'B'  Icon ('.' Icon)?                 -> DataEditDescr
  'Z'  Icon ('.' Icon)?                 -> DataEditDescr
  'F'  Icon '.' Icon                    -> DataEditDescr
  'E'  Icon '.' Icon ('E' Icon)?        -> DataEditDescr
  'EN' Icon '.' Icon ('E' Icon)?        -> DataEditDescr
  'ES' Icon '.' Icon ('E' Icon)?        -> DataEditDescr
  'G'  Icon '.' Icon ('E' Icon)?        -> DataEditDescr
  'L'  Icon                             -> DataEditDescr
  'A'  Icon?                            -> DataEditDescr
  'D'  Icon '.' Icon                    -> DataEditDescr

%%R1010,R1011
  PositionEditDescr                     -> ControlEditDescr
  Icon? '/'                             -> ControlEditDescr
  ':'                                   -> ControlEditDescr
  SignEditDescr                         -> ControlEditDescr
  ('-'|'+')? Icon 'P' (Icon? DataEditDescr)?    -> ControlEditDescr
  BlankInterpEditDescr                  -> ControlEditDescr

%%R1012, R1013
  'T'  Icon                             -> PositionEditDescr
  'TL' Icon                             -> PositionEditDescr
  'TR' Icon                             -> PositionEditDescr
  Icon 'X'                              -> PositionEditDescr

%%R1014
  'S'                                   -> SignEditDescr
  'SP'                                  -> SignEditDescr
  'SS'                                  -> SignEditDescr

%%R1015
  'BN'                                  -> BlankInterpEditDescr
  'BZ'                                  -> BlankInterpEditDescr

%%R1016
  Scon                                  -> CharStringEditDescr
  Icon 'H' Character+                   -> CharStringEditDescr
%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 section R11xx ProgramUnits
%%%%%%%%%

module languages/fortran/syntax/R1100ProgramUnits

imports
  languages/fortran/syntax/FortranLex
  languages/fortran/syntax/Fortran90

exports

sorts
  BlockDataBody BlockDataBodyConstruct BlockDataName BlockDataStmt BlockDataSubprogram EndBlockDataStmt
  EndModuleStmt EndProgramStmt Module ModuleBody ModuleName ModuleStmt
  Only OnlyList ProgramStmt Rename RenameList UseName UseStmt

context-free syntax

%%R1102
```

```
  LblDef 'program' ProgramName EOS              -> ProgramStmt

%%R1103
  LblDef 'end' EOS                              -> EndProgramStmt
  LblDef 'end' 'program' EndName? EOS           -> EndProgramStmt

%%R1104
  ModuleStmt ModuleBody EndModuleStmt           -> Module
  ModuleStmt EndModuleStmt                      -> Module
  SpecificationPartConstruct                    -> ModuleBody
  ModuleSubprogramPartConstruct                 -> ModuleBody
  ModuleBody SpecificationPartConstruct         -> ModuleBody
  ModuleBody ModuleSubprogramPartConstruct      -> ModuleBody

%%R1105
  LblDef 'module' ModuleName EOS                -> ModuleStmt
%% alias
  Ident  -> ModuleName

%%R1106
  LblDef 'end' EOS                              -> EndModuleStmt
  LblDef 'end' 'module' EndName? EOS            -> EndModuleStmt

%%R1107
  LblDef 'use' Name ( ',' RenameList )? EOS             -> UseStmt
  LblDef 'use' Name ',' 'only' ':' OnlyList? EOS        -> UseStmt
  {Rename ","}+                                         -> RenameList
  {Only  ","}+                                          -> OnlyList

%%R1108
  Ident '=>' UseName                            -> Rename

%% aliases
  Ident  -> UseName
  Ident  -> BlockDataName


%%R1109
  GenericSpec                                   -> Only
  ( Ident '=>' )? UseName                       -> Only

%%R1110
  BlockDataStmt BlockDataBody EndBlockDataStmt          -> BlockDataSubprogram
  BlockDataStmt EndBlockDataStmt                        -> BlockDataSubprogram

  BlockDataBodyConstruct+       -> BlockDataBody
  SpecificationPartConstruct    -> BlockDataBodyConstruct

%%R1111
  LblDef 'block' 'data' BlockDataName? EOS      -> BlockDataStmt

%%R1112
  LblDef 'end' 'block' 'data' EndName? EOS      -> EndBlockDataStmt
  LblDef 'end' EOS                              -> EndBlockDataStmt

%%%%%%%%%%
%% Fortran ISO/IEC 1539:1991 section 12xx Procedures section
%%%%%%%%%%

module languages/fortran/syntax/R1200Procedures

imports
  languages/fortran/syntax/FortranLex
  languages/fortran/syntax/Fortran90

exports

sorts

  ActualArg CallStmt ContainsStmt EndFunctionStmt EndInterfaceStmt
  EndSubroutineStmt EntryStmt ExternalStmt FunctionArg FunctionArgList
  FunctionInterfaceRange FunctionPar FunctionParList FunctionPrefix FunctionRange
  FunctionReference FunctionStmt FunctionSubprogram GenericSpec InterfaceBlock
  InterfaceBlockPart InterfaceBody InterfaceStmt IntrinsicStmt ModuleProcedureStmt
  ProcedureName ProcedureNameList ReturnStmt SFDummyArgNameList
  %% StmtFunctionStmt obsolescant/deleted
```

```
  SubprogramInterfaceBody SubroutineInterfaceRange SubroutinePar SubroutineParList SubroutineRange
  SubroutineStmt SubroutineSubprogram

context-free syntax

%%R1201 %% note: iso says interfacebody allways precedes moduleprocstmt.
  InterfaceStmt InterfaceBlockPart+ EndInterfaceStmt    -> InterfaceBlock

  InterfaceBody                                 -> InterfaceBlockPart
  ModuleProcedureStmt                           -> InterfaceBlockPart

%%R1202
  LblDef 'interface' GenericName EOS            -> InterfaceStmt
  LblDef 'interface' GenericSpec EOS            -> InterfaceStmt
  LblDef 'interface' EOS                        -> InterfaceStmt

%%R1203
  LblDef 'end' 'interface' EOS                  -> EndInterfaceStmt

%%R1204
  LblDef FunctionPrefix FunctionName FunctionInterfaceRange     -> InterfaceBody
  LblDef 'subroutine' SubroutineName SubroutineInterfaceRange   -> InterfaceBody

  FunctionParList   EOS SubprogramInterfaceBody EndFunctionStmt         -> FunctionInterfaceRange
  FunctionParList   EOS EndFunctionStmt                                 -> FunctionInterfaceRange

  SubroutineParList EOS SubprogramInterfaceBody EndSubroutineStmt       -> SubroutineInterfaceRange
  SubroutineParList EOS EndSubroutineStmt                               -> SubroutineInterfaceRange

  SpecificationPartConstruct                            -> SubprogramInterfaceBody
  SubprogramInterfaceBody SpecificationPartConstruct    -> SubprogramInterfaceBody

%%R1205
  LblDef 'module' 'procedure' ProcedureNameList EOS     -> ModuleProcedureStmt
  {ProcedureName ","}+                                  -> ProcedureNameList
  Ident                                                 -> ProcedureName

%%R1206
  'operator' '(' DefinedOperator ')'            -> GenericSpec
  'assignment' '(' '=' ')'                      -> GenericSpec

%%R1207
  LblDef 'external' {ExternalName ","}+ EOS     -> ExternalStmt

%%R1208
  LblDef 'intrinsic' {IntrinsicProcedureName ","}+ EOS  -> IntrinsicStmt

%%R1209
  Name '(' FunctionArgList? ')'                         -> FunctionReference

%%R1210
  LblDef 'call' SubroutineNameUse EOS                           -> CallStmt
  LblDef 'call' SubroutineNameUse '(' {ActualArg ","}* ')' EOS  -> CallStmt

%%R1213
  (Name '=')? Expr                      -> ActualArg
  (Name '=')? '*' LblRef                -> ActualArg

%%R1211 [ keyword = ] actual-arg
  FunctionArg                                   -> FunctionArgList
  FunctionArgList ',' FunctionArg               -> FunctionArgList
  SectionSubscriptList ',' FunctionArg          -> FunctionArgList

  Name '=' Expr                                 -> FunctionArg

%%R1212 keyword is dummy-arg-name
%% see 1211

%%R1214 alt-return-spec is '*' label
%% see 1213

%%R1215
  LblDef FunctionPrefix FunctionName FunctionRange      -> FunctionSubprogram
  FunctionParList EOS Body? EndFunctionStmt             -> FunctionRange

%%R1215
```

```
  FunctionParList 'result' '(' Name ')' EOS InternalSubProgPart EndFunctionStmt         -> FunctionRange
  FunctionParList 'result' '(' Name ')' EOS Body EndFunctionStmt                        -> FunctionRange
  FunctionParList 'result' '(' Name ')' EOS EndFunctionStmt                             -> FunctionRange
  FunctionParList EOS InternalSubProgPart EndFunctionStmt                               -> FunctionRange

%%R1216
%%R1217
  'recursive' 'function'                -> FunctionPrefix
  'recursive' TypeSpec 'function'       -> FunctionPrefix
  TypeSpec 'recursive' 'function'       -> FunctionPrefix

%%R1218
%% endfunction (without spaces) is allowed
  LblDef 'end' EOS                              -> EndFunctionStmt
  LblDef 'end' 'function' EndName? EOS          -> EndFunctionStmt

%%R1219
%%R1220
  LblDef 'recursive'? 'subroutine' SubroutineName SubroutineRange       -> SubroutineSubprogram
  SubroutineParList? EOS Body? EndSubroutineStmt                        -> SubroutineRange
  SubroutineParList  EOS InternalSubProgPart EndSubroutineStmt          -> SubroutineRange
%% split subroutineStmt/subroutineSubprogram needed? (ELI legacy)
  LblDef 'subroutine' Name SubroutineParList? EOS                       -> SubroutineStmt
   '(' {SubroutinePar ","}* ')'                                         -> SubroutineParList

%%R1221
   DummyArgName         -> SubroutinePar
   '*'                  -> SubroutinePar

%%/* Must be split on semantic grounds, due to the different scopes for the
   LblDef FunctionPrefix Name FunctionParList? EOS      -> FunctionStmt
   TypeSpec? 'function'                                 -> FunctionPrefix

   '(' {FunctionPar ","}* ')'                           -> FunctionParList
   DummyArgName                                         -> FunctionPar

%%R1222
  LblDef 'end' 'subroutine' EndName? EOS        -> EndSubroutineStmt
  LblDef 'end' EOS                              -> EndSubroutineStmt

%%R1223
  LblDef 'entry' EntryName SubroutineParList EOS                        -> EntryStmt
  LblDef 'entry' EntryName SubroutineParList 'result' '(' Name ')' EOS  -> EntryStmt

%%R1224
  LblDef 'return' Expr? EOS             -> ReturnStmt

%%R1225
  LblDef 'contains' EOS                 -> ContainsStmt

%%-R1226
%% statement-functions are obsolete in fortran 95.
%% The following rule causes ambiguity with assignment statements like
%% FOO(NOARG) = IN2-IN3
%% If you're ok with ambig rules then uncomment the next line.
%% LblDef  Name '(' SFDummyArgNameList? ')' '=' Expr EOS        -> StmtFunctionStmt

  {SFDummyArgName ","}+                 -> SFDummyArgNameList
```