Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
INE5426 - Construção de Compiladores

Analisador Sintático

Alunos:
Caetano Colin Torres
Alek Frohlich
Nícolas Goeldner

Florianópolis, 2021

Sumário

1.1 Conversão da Gramática para convencional **3**
Tabela 1.1 Gramática na Forma Convencional; 4

1.2. Recursão a Esquerda **5**
Tabela 1.2. Gramática Livre de Recursão à Esquerda. 11

1.3. Fatoração à esquerda **11**
Tabela 1.3. Gramática Fatorada à Esquerda. 14

1.4. Gramática LL(1) **15**
Tabela 1.4.1 Conjuntos First e Follow antes da gramática transformada para LL(1). 18
Tabela 1.4.2 Novas produções da gramática, resolvendo conflitos. 19

1.5 Descrição da Ferramenta e Tabela de Símbolos **19**

1.1 Conversão da Gramática para convencional

Nesta seção iremos transformar a gramática da forma BNF para convencional. Para isso, deve-se remover os parênteses das regras, criar regras que omitam os operadores de expressão regular (+,*,?), Com isso temos a seguinte gramática:

Head	**Body**
PROGRAM	STATEMENT | FUNCLIST | ε
FUNCLIST	FUNCDEF FUNCLIST | FUNCDEF
FUNCDEF	def ident (PARAMLIST) { STATELIST }
PARAMLIST	int ident, PARAMLIST | float ident, PARAMLIST | string ident, PARAMLIST | int ident | float ident | string ident | ε
STATEMENT	VARDECL ; | ATRIBSTAT ; | PRINTSTAT ; | READSTAT ; | return ; | IFSTAT | FORSTAT | { STATELIST } | break ; | ;
VARDECL	int ident ARRAYLISTDECL | float ident ARRAYLISTDECL | string ident ARRAYLISTDECL
ATRIBSTAT	LVALUE = EXPRESSION | LVALUE = ALLOCEXPRESSION | LVALUE = FUNCCALL
FUNCCALL	ident(PARAMLISTCALL)
PARAMLISTCALL	ident, PARAMLISTCALL | ident | ε
PRINTSTAT	print EXPRESSION
READSTAT	read LVALUE
IFSTAT	if(EXPRESSION) { STATELIST } ELSESTAT
ELSESTAT	else { STATELIST } | ε
FORSTAT	for (ATRIBSTAT ; EXPRESSION ; ATRIBSTAT) STATEMENT
STATELIST	STATEMENT STATELIST | STATEMENT

ALLOCEXPRESSION	new int [NUMEXPRESSION] ARRAYLISTEXP | new float [NUMEXPRESSION] ARRAYLISTEXP | new string [NUMEXPRESSION] ARRAYLISTEXP
EXPRESSION	NUMEXPRESSION < NUMEXPRESSION | NUMEXPRESSION > NUMEXPRESSION | NUMEXPRESSION <= NUMEXPRESSION | NUMEXPRESSION >= NUMEXPRESSION | NUMEXPRESSION == NUMEXPRESSION | NUMEXPRESSION != NUMEXPRESSION | NUMEXPRESSION
NUMEXPRESSION	NUMEXPRESSION + TERM | NUMEXPRESSION - TERM | TERM
TERM	TERM * UNARYEXPR | TERM / UNARYEXPR | TERM % UNARYEXPR | UNARYEXPR
UNARYEXPR	+ FACTOR | - FACTOR | FACTOR
FACTOR	int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE	ident ARRAYLISTEXP
ARRAYLISTDECL	ARRAYLISTDECL [int_constant] | ε
ARRAYLISTEXP	ARRAYLISTEXP [NUMEXPRESSION] | ε

Tabela 1.1 Gramática na Forma Convencional;

1.2. Recursão a Esquerda

Para saber se a gramática é recursiva à esquerda devemos olhar cada produção e identificar se ocorre recursão à esquerda.

PROGRAM	STATEMENT | FUNCLIST | ε

A regra acima não possui recursão à esquerda, porque PROGRAM só produz o corpo da produção e não aparece mais em outras derivações.

FUNCLIST	FUNCDEF FUNCLIST | FUNCDEF

A regra acima sempre terá FUNCDEF mais à esquerda, e após analisar a produção de FUNCDEF percebe-se que ela sempre inicia com um terminal mais à esquerda, ou seja, FUNCLIST nunca estará mais à esquerda nas derivações e portanto não é uma produção recursiva à esquerda.

FUNCDEF	def ident (PARAMLIST) { STATELIST }

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

PARAMLIST	int ident, PARAMLIST | float ident, PARAMLIST | string ident, PARAMLIST | int ident | float ident | string ident | ε

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

STATEMENT	VARDECL ; | ATRIBSTAT ; | PRINTSTAT ; | READSTAT ; | return ; | IFSTAT | FORSTAT | { STATELIST } | break ; | ;

Para a regra acima, temos as 3 produções `{ STATELIST } | break ; | return; | ;` que iniciam com algum terminal mais à esquerda. Deve-se analisar agora as outras produções, a produção VARDECL sempre começa com um terminal mais à esquerda, portanto não teremos problema quanto a ela. ATRIBSTAT produz LVALUE mais à esquerda que por sua vez produz um terminal mais à esquerda. Quanto às outras regras, PRINTSTAT, READSTAT, FORSTAT, IFSTAT elas sempre começam produzindo um terminal mais à esquerda.

VARDECL	int ident ARRAYLISTDECL | float ident ARRAYLISTDECL | string ident ARRAYLISTDECL

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

ATRIBSTAT	LVALUE = EXPRESSION | LVALUE = ALLOCEXPRESSION | LVALUE = FUNCCALL

Para a regra ATRIBSTAT, ela começa sempre produzindo a regra LVALUE mais à esquerda que por sua vez produz sempre um terminal mais à esquerda, portanto não é recursiva à esquerda.

FUNCCALL	ident(PARAMLISTCALL)

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

PARAMLISTCALL	ident, PARAMLISTCALL | ident | ε

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

PRINTSTAT	print EXPRESSION

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

READSTAT	read LVALUE

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

IFSTAT	if(EXPRESSION) { STATELIST } ELSESTAT

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

ELSESTAT	else { STATELIST } | ε

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

FORSTAT	for (ATRIBSTAT ; EXPRESSION ; ATRIBSTAT) STATEMENT

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

STATELIST	STATEMENT STATELIST | STATEMENT

Para a regra STATELIST, ela sempre começa produzindo STATEMENT mais à esquerda. Como já analisado antes, essa regra tem 3 produções produzindo um terminal mais à esquerda e as outras regras com não terminais começam sempre produzindo um terminal mais à esquerda. VARDECL sempre começa com um terminal mais à esquerda, portanto não teremos problema quanto a ela. ATRIBSTAT produz LVALUE mais à esquerda que por sua vez produz um terminal mais à esquerda. Quanto às outras regras, PRINTSTAT, READSTAT, FORSTAT, IFSTAT elas sempre começam produzindo um terminal mais à esquerda. Desse modo, STATELIST não é recursivo à esquerda.

ALLOCEXPRESSION	new int [NUMEXPRESSION] ARRAYLISTEXP | new float [NUMEXPRESSION] ARRAYLISTEXP | new string [NUMEXPRESSION] ARRAYLISTEXP

A regra acima sempre inicia com um terminal mais à esquerda, portanto não é recursiva à esquerda.

EXPRESSION	NUMEXPRESSION < NUMEXPRESSION | NUMEXPRESSION > NUMEXPRESSION | NUMEXPRESSION <= NUMEXPRESSION | NUMEXPRESSION >= NUMEXPRESSION | NUMEXPRESSION == NUMEXPRESSION | NUMEXPRESSION != NUMEXPRESSION | NUMEXPRESSION

Analisando a regra acima, vemos que ela sempre deriva NUMEXPRESSION mais a esquerda, e analisando NUMEXPRESSION já percebemos uma recursão a esquerda, portanto devemos resolver isso.

NUMEXPRESSION	NUMEXPRESSION + TERM | NUMEXPRESSION - TERM | TERM

Analisando agora NUMEXPRESSION, percebemos que ela é recursiva à esquerda, portanto adaptamos sua produção para:

NUMEXPRESSION	TERM NUMEXPRESSION'
NUMEXPRESSION’	+ TERM NUMEXPRESSION' | - TERM NUMEXPRESSION' | ε

Percebemos também que TERM é recursivo a esquerda e adaptamos a produção

TERM	TERM * UNARYEXPR | TERM / UNARYEXPR | TERM % UNARYEXPR | UNARYEXPR

para:

TERM	UNARYEXPR TERM'
TERM’	* UNARYEXPR TERM' | / UNARYEXPR TERM' | % UNARYEXPR TERM' | ε

Analisando agora as produções UNARYEXPR, vemos que ela deriva FACTOR e FACTOR deriva LVALUE que por sua vez não é recursivo à esquerda, pois deriva à esquerda um terminal. Assim, percebemos, também, que FACTOR e UNARYEXPR não são recursivos à esquerda.

UNARYEXPR	+ FACTOR | - FACTOR | FACTOR
FACTOR	int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE	ident ARRAYLISTEXP

Olhando a produção ARRAYLISTEXP, percebemos recursão à esquerda, então adaptamos a produção:

ARRAYLISTEXP	ARRAYLISTEXP [NUMEXPRESSION] | ε

para:

ARRAYLISTEXP	ARRAYLISTEXP'
ARRAYLISTEXP’	[NUMEXPRESSION] ARRAYLISTEXP' | ε

Desse modo resolvemos as recursões embutidas em NUMEXPRESSION.

Temos ainda uma produção que não foi analisada e é recursiva à esquerda, que é

ARRAYLISTDECL	ARRAYLISTDECL [int_constant] | ε

Portanto adaptamos ela para:

ARRAYLISTDECL	ARRAYLISTDECL’
ARRAYLISTDECL’	[int_constant] ARRAYLISTDECL’ | ε

Com isso, resolvemos as recursões a esquerda em ARRAYLISTDECL, ARRAYLISTEXP, TERM e NUMEXPRESSION. Desse modo, a gramática está livre de recursão à esquerda. Agora, a gramática completa está da seguinte forma:

Head	**Body**
PROGRAM	STATEMENT | FUNCLIST | ε
FUNCLIST	FUNCDEF FUNCLIST | FUNCDEF
FUNCDEF	def ident (PARAMLIST) { STATELIST }
PARAMLIST	int ident, PARAMLIST | float ident, PARAMLIST | string ident, PARAMLIST | int ident | float ident | string ident | ε
STATEMENT	VARDECL ; | ATRIBSTAT ; | PRINTSTAT ; | READSTAT ; | return ; | IFSTAT | FORSTAT | { STATELIST } | break ; | ;
VARDECL	int ident ARRAYLISTDECL | float ident ARRAYLISTDECL | string

	ident ARRAYLISTDECL
ATRIBSTAT	LVALUE = EXPRESSION | LVALUE = ALLOCEXPRESSION | LVALUE = FUNCCALL
FUNCCALL	ident(PARAMLISTCALL)
PARAMLISTCALL	ident, PARAMLISTCALL | ident | ε
PRINTSTAT	print EXPRESSION
READSTAT	read LVALUE
IFSTAT	if(EXPRESSION) { STATELIST } ELSESTAT
ELSESTAT	else { STATELIST } | ε
FORSTAT	for (ATRIBSTAT ; EXPRESSION ; ATRIBSTAT) STATEMENT
STATELIST	STATEMENT STATELIST | STATEMENT
ALLOCEXPRESSION	new int [NUMEXPRESSION] ARRAYLISTEXP | new float [NUMEXPRESSION] ARRAYLISTEXP | new string [NUMEXPRESSION] ARRAYLISTEXP
EXPRESSION	NUMEXPRESSION < NUMEXPRESSION | NUMEXPRESSION > NUMEXPRESSION | NUMEXPRESSION <= NUMEXPRESSION | NUMEXPRESSION >= NUMEXPRESSION | NUMEXPRESSION == NUMEXPRESSION | NUMEXPRESSION != NUMEXPRESSION | NUMEXPRESSION
NUMEXPRESSION	TERM NUMEXPRESSION'
NUMEXPRESSION’	+ TERM NUMEXPRESSION' | - TERM NUMEXPRESSION'| ε
TERM	UNARYEXPR TERM'
TERM’	* UNARYEXPR TERM' | / UNARYEXPR TERM' | % UNARYEXPR TERM' | ε
UNARYEXPR	+ FACTOR | - FACTOR | FACTOR
FACTOR	int_constant | float_constant |

	string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE	ident ARRAYLISTEXP
ARRAYLISTDECL	ARRAYLISTDECL'
ARRAYLISTDECL’	[int constant] ARRAYLISTDECL' | ε
ARRAYLISTEXP	ARRAYLISTEXP'
ARRAYLISTEXP’	[NUMEXPRESSION] ARRAYLISTEXP' | ε

Tabela 1.2. Gramática Livre de Recursão à Esquerda.

1.3. Fatoração à esquerda

Uma gramática não é fatorada à esquerda se ela possui alguma regra em que pelo menos duas de suas produções tenham algum prefixo em comum. Para fatorar a gramática, basta manter a produção do prefixo em uma única produção e criar uma nova regra com suas possíveis derivações, dessa forma não restando prefixos em comum em nenhuma regra.

Ao analisar a Tabela 1.2. vemos que apenas as produções FUNCLIST, PARAMLIST, ATRIBSTAT, PARAMLISTCALL, STATELIST, ALLOCEXPRESSION e EXPRESSION precisam ser fatoradas à esquerda porque possuem pelo menos duas de suas derivações com prefixos em comum: Fatorando essas regras obtemos as seguintes novas regras:

Para FUNCLIST:

FUNCLIST	FUNCDEF FUNCLIST’
FUNCLIST’	FUNCLIST | ε

Para PARAMLIST:

PARAMLIST	string ident PARAMLIST' | float ident PARAMLIST' | int ident PARAMLIST' | ε
PARAMLIST’	, PARAMLIST | ε

Para ATRIBSTAT:

ATRIBSTAT	LVALUE = ATRIBSTAT'

ATRIBSTAT’	`EXPRESSION | ALLOCEXPRESSION | FUNCCALL`

Para PARAMLISTCALL:

PARAMLISTCALL	`ident PARAMLISTCALL' | ε`
PARAMLISTCALL’	`, PARAMLISTCALL | ε`

Para STATELIST:

STATELIST	`STATEMENT STATELIST'`
STATELIST’	`STATELIST | ε`

Para ALLOCEXPRESSION:

ALLOCEXPRESSION	`new ALLOCEXPRESSION'`
ALLOCEXPRESSION’	`int [ NUMEXPRESSION ] ARRAYLISTEXP | float [ NUMEXPRESSION ] ARRAYLISTEXP | string [ NUMEXPRESSION ] ARRAYLISTEXP`

Para EXPRESSION:

EXPRESSION	`NUMEXPRESSION EXPRESSION'`
EXPRESSION’	`< NUMEXPRESSION | > NUMEXPRESSION | <= NUMEXPRESSION | >= NUMEXPRESSION | == NUMEXPRESSION | != NUMEXPRESSION | ε`

Com isso temos:

Head	**Body**
PROGRAM	STATEMENT | FUNCLIST | ε
FUNCLIST	FUNCDEF FUNCLIST’
FUNCLIST’	FUNCLIST | ε
FUNCDEF	def ident (PARAMLIST) { STATELIST }
PARAMLIST	string ident PARAMLIST' | float ident PARAMLIST' | int ident PARAMLIST' | ε
PARAMLIST’	, PARAMLIST | ε
STATEMENT	VARDECL ; | ATRIBSTAT ; | PRINTSTAT ; | READSTAT ; | return ; | IFSTAT | FORSTAT | { STATELIST } | break ; | ;
VARDECL	int ident ARRAYLISTDECL | float ident ARRAYLISTDECL | string ident ARRAYLISTDECL
ATRIBSTAT	LVALUE = ATRIBSTAT'
ATRIBSTAT’	EXPRESSION | ALLOCEXPRESSION | FUNCCALL
FUNCCALL	ident(PARAMLISTCALL)
PARAMLISTCALL	ident PARAMLISTCALL' | ε
PARAMLISTCALL’	, PARAMLISTCALL | ε
PRINTSTAT	print EXPRESSION
READSTAT	read LVALUE
IFSTAT	if(EXPRESSION) { STATELIST } ELSESTAT
ELSESTAT	else { STATELIST } | ε
FORSTAT	for (ATRIBSTAT ; EXPRESSION ; ATRIBSTAT) STATEMENT
STATELIST	STATEMENT STATELIST'
STATELIST’	STATELIST | ε

ALLOCEXPRESSION	`new ALLOCEXPRESSION'`
ALLOCEXPRESSION’	`int [ NUMEXPRESSION ] ARRAYLISTEXP | float [ NUMEXPRESSION ] ARRAYLISTEXP | string [ NUMEXPRESSION ] ARRAYLISTEXP`
EXPRESSION	`NUMEXPRESSION EXPRESSION'`
EXPRESSION’	`< NUMEXPRESSION | > NUMEXPRESSION | <= NUMEXPRESSION | >= NUMEXPRESSION | == NUMEXPRESSION | != NUMEXPRESSION | ε`
NUMEXPRESSION	`TERM NUMEXPRESSION'`
NUMEXPRESSION’	`+ TERM NUMEXPRESSION' | - TERM NUMEXPRESSION'| ε`
TERM	`UNARYEXPR TERM'`
TERM’	`* UNARYEXPR TERM' | / UNARYEXPR TERM' | % UNARYEXPR TERM' | ε`
UNARYEXPR	`+ FACTOR | - FACTOR | FACTOR`
FACTOR	`int constant | float constant | string constant | null | LVALUE | ( NUMEXPRESSION )`
LVALUE	`ident ARRAYLISTEXP`
ARRAYLISTDECL	`ARRAYLISTDECL'`
ARRAYLISTDECL’	`[ int constant ] ARRAYLISTDECL' | ε`
ARRAYLISTEXP	`ARRAYLISTEXP'`
ARRAYLISTEXP’	`[ NUMEXPRESSION ] ARRAYLISTEXP' | ε`

Tabela 1.3. Gramática Fatorada à Esquerda.

1.4. Gramática LL(1)

LL(1) significa que a varredura será da esquerda para direita, que a derivação será mais à esquerda e um token na entrada deve ser lido por vez. A gramática está em LL(1) se sempre que tivermos duas produções distintas **A -> a | b**, as seguintes condições são satisfeitas:

- first(a)∩first(b) = Ø
- Se b deriva para ε com 0 ou mais derivações então first(a)∩follow(A) = Ø
- Se a deriva para ε com 0 ou mais derivações então first(b)∩follow(A) = Ø

Basta olhar as produções uma por uma e verificar se as condições são satisfeitas. Após analisar as produções junto dos conjuntos First/Follow listados na tabela 1.3.1 abaixo percebe-se que ocorrem conflitos em ATRIBSTAT'.

Cabeça da Produção	Anulável	Conjunto First	Conjunto Follow
ALLOCEXPRESSION	false	new	) ;
ALLOCEXPRESSION'	false	float int string	) ;
ARRAYLISTDECL	true	[ϵ	;
ARRAYLISTDECL'	true	[ϵ	;
ARRAYLISTEXP	true	[ϵ	!= %) * + - / ; < <= = == > >=]
ARRAYLISTEXP'	true	[ϵ	!= %) * + - / ; < <= = == > >=]
ATRIBSTAT	false	ident	) ;
ATRIBSTAT'	false	(+ - float_constant ident int_constant new null string_constant	) ;

ELSESTAT	true	else ϵ	$; break float for ident if(int print read return string { }
EXPRESSION	false	(+ - float_constant ident int_constant null string_constant	) ;
EXPRESSION'	true	!= < <= == > >= ϵ	) ;
FACTOR	false	(float_constant ident int_constant null string_constant	!= %) * + - / ; < <= == > >=]
FORSTAT	false	for	$; break float for ident if(int print read return string { }
FUNCCALL	false	ident	) ;
FUNCDEF	false	def	$ def
FUNCLIST	false	def	$
FUNCLIST'	true	def ϵ	$
IFSTAT	false	if(	$; break float for ident if(int print read return string { }
LVALUE	false	ident	!= %) * + - / ; < <= = == > >=]

NUMEXPRESSION	false	(+ - float_constant ident int_constant null string_constant	!=) ; < <= == > >=]
NUMEXPRESSION'	true	+ - ϵ	!=) ; < <= == > >=]
PARAMLIST	true	float int string ϵ	)
PARAMLIST'	true	, ϵ	)
PARAMLISTCALL	true	ident ϵ	)
PARAMLISTCALL'	true	, ϵ	)
PRINTSTAT	false	print	;
PROGRAM	true	; break def float for ident if(int print read return string { ϵ	$
READSTAT	false	read	;
STATELIST	false	; break float for ident if(int print read return string {	}
STATELIST'	true	; break float for ident if(int print read return string { ϵ	}

STATEMENT	false	; break float for ident if(int print read return string {	$; break float for ident if(int print read return string { }
TERM	false	(+ - float_constant ident int_constant null string_constant	!=) + - ; < <= == > >=]
TERM'	true	% * / ϵ	!=) + - ; < <= == > >=]
UNARYEXPR	false	(+ - float_constant ident int_constant null string_constant	!= %) * + - / ; < <= == > >=]
VARDECL	false	float int string	;

Tabela 1.4.1 Conjuntos First e Follow antes da gramática transformada para LL(1).

Analisando a produção ATRIBSTAT', percebe-se que first(FUNCCALL)∩first(EXPRESSION) ≠ Ø.

ATRIBSTAT’	EXPRESSION | ALLOCEXPRESSION | FUNCCALL

Cabeça da Produção	**Anulável**	**Conjunto First**	**Conjunto Follow**
FUNCCALL	false	ident	) ;
EXPRESSION	false	(+ - float_constant ident int_constant null string_constant	) ;

Resolvendo esse conflito através das novas produções listadas abaixo nossa gramática será LL(1).

ATRIBSTAT'	+ FACTOR TERM' NUMEXPRESSION' EXPRESSION' | - FACTOR TERM' NUMEXPRESSION' EXPRESSION' | int_constant TERM' NUMEXPRESSION' EXPRESSION' | float_constant TERM' NUMEXPRESSION' EXPRESSION' | string_constant TERM' NUMEXPRESSION' EXPRESSION' | null TERM' NUMEXPRESSION' EXPRESSION' | (NUMEXPRESSION) TERM' NUMEXPRESSION' EXPRESSION' | ALLOCEXPRESSION | ident ATRIBSTAT''
ATRIBSTAT''	(PARAMLISTCALL) | ARRAYLISTEXP TERM' NUMEXPRESSION' EXPRESSION'

Tabela 1.4.2 Nova produção da gramática, resolvendo conflitos.

1.5 Descrição da Ferramenta e Tabela de Símbolos

A Tabela de símbolos é uma estrutura de dados utilizada para o armazenamento de informações de identificadores, onde cada identificador é associado com uma informação relacionada a declaração ou ocorrência no código fonte - como tipo e layout de armazenamento. Basicamente a tabela armazena informações relacionadas a símbolos de entrada. No caso do nosso trabalho, a tabela de símbolos foi implementada na fase de análise sintática.

Na nossa tabela de símbolos utilizamos uma stack cujos elementos são os vários escopos do código. Assim que abrimos um escopo, ele é colocado na stack e, quando o fechamos, damos pop. A ideia é que conforme vamos entrando em escopos mais profundos, vamos aumentando a stack, mantendo os escopos pai no fundo até que eles sejam fechados, com isso podemos obter informações de escopos pai. Por enquanto, estamos armazenando apenas o tipo do identificador, podendo este ser: INT, FLOAT, STRING, FUNCTION. Os símbolos armazenados são funções e variáveis.

Ilustramos esse processo de aninhamento de escopos com auxílio da figura 7, que contém o trecho da gramática responsável pela definição de funções. Começamos a declaração de uma função, colocando seu nome no escopo global e então abrimos o escopo de seus parâmetros; cada parâmetro é colocado na tabela; depois abrimos o escopo para o corpo da função, ou seja, o que está entre as chaves sem fechar o escopo dos parâmetros; depois que termina a função (fecha as chaves), fechamos o escopo do corpo e depois fechamos o escopo dos parâmetros, terminando a definição função.

```
funclist: funcdef funclist | funcdef;
funcdef: DEF IDENT                    { if(!check_put(std::string($2), SymType::T_FUNC)) YYABORT; Env::open_scope();}
                '(' paramlist ')' '{' { Env::open_scope(); }
                    statelist '}'     { Env::close_scope(); Env::close_scope(); }
;

paramlist: type IDENT { if(!check_put(std::string($2), $1)) YYABORT; } ',' paramlist
        |  type IDENT { if(!check_put(std::string($2), $1)) YYABORT; }
        |  %empty
;
type: INT { $$ = 0; } | FLOAT { $$ = 1; } | STRING { $$ = 2; };
```

Figura 7. Exemplo de resolução de escopo.

Usamos a ferramenta ***flex*** para gerar um analisador léxico, e posteriormente usamos o Bison para gerar um analisador sintático LALR(1). ***flex*** lê de uma stream de entrada especificando o formato do analisador léxico (.l) contendo pares de definições regulares e fragmentos de código (*rules*), e gera como saída um arquivo fonte na linguagem C que define uma rotina ***yylex()***. Esse arquivo é compilado e ligado a arquivos .cpp utilizando ***g++*** para produzir um executável. Quando o executável é rodado, ele analisa a entrada para ocorrências das expressões regulares e executa ações (trechos de código) relacionadas a cada padrão, ao passo que este é encontrado.

Um exemplo de como usar ***flex*** é o input a seguir que especifica um scanner que quando encontra a string 'username' substitui pelo nome do usuário logado.

```
%%
username    printf( "%s", getlogin() );
```

Por padrão, qualquer texto que não tiver um ***match*** pelo scanner ***flex*** é copiado para o output. Assim, o efeito do scanner é copiar o arquivo entrada para a saída com cada ocorrência de 'username' expandida. No exemplo acima, só existe uma regra: 'username' é o padrão e 'printf' é a ação. O símbolo %% marca o início das regras.

Um arquivo de entrada ***flex*** consiste de três seções separadas por uma linha contendo ***%%***.

```
definitions
%%
rules
%%
user code
```

A seção de definições contém declarações de nome simples para simplificar a especificação do analisador. Definições de nome tem o formato ***name definition***, Um exemplo é:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

Que define 'DIGIT' como uma expressão regular que marca ocorrências de um dígito único, e 'ID' é a expressão regular que marca ocorrências de uma letra seguida de um ou mais letras/dígitos.

A seção de regras contém uma série de regras na forma ***pattern action***; pode-se definir variáveis nessa seção que são locais para a rotina do analisador. Na seção de user code, o código é apenas copiado para ***lex.yy.c*** e pode ser usado para rotinas que são chamadas ou chamam o analisador.

A saída do ***flex*** é o arquivo ***lex.yy.c***, que contém a rotina de análise léxica ***yylex()***, um número de tabelas usadas pelos tokens e um número de rotinas e macros auxiliares. Por padrão, ***yylex()*** é declarado como:

```
int yylex()
    {
    ... various definitions and the actions in here ...
    }
```

Toda vez que ***yylex()*** é chamada, a entrada é examinada até que outra ocorrência de um dos padrões seja encontrada. Quando um padrão é reconhecido, ***yylex()*** retorna um identificador avisando qual token foi encontrado. Esse identificador pode ser qualquer inteiro, porém, como utilizamos o ***flex*** em conjunto com o ***Bison*** (gerador de analisador sintático), seguimos a convenção em que os identificadores vem de uma enumeração (**yytokentype**); esta enumeração é definida pelo arquivo **xpp.tab.h**, que é gerado pela ferramenta ***Bison***, conforme Figura 1 vista na seção 1. O valor do token é armazenado na global **yylval**.

O ***Bison*** recebe como entrada a especificação de um analisador sintático/semântico[1] (.y) contendo um esquema de tradução dirigido pela sintáxe dado por uma gramática LALR(1) anotada com ações semânticas. Esta entrada é usada para gerar um analisador sintático/semântico, cuja rotina principal é **yyparse()**, e um cabeçalho definindo **yytokentype**, ambos baseados na linguagem C. Quando as saídas **xpp.tab.c** e **xpp.tab.h** são compiladas com o analisador léxico gerado pelo ***flex***, obtém-se um executável que implementa o esquema de tradução que fora recebido como entrada. Quando executado, o programa realiza análise léxica até encontrar um token. Tendo encontrado um token, este é entregue ao analisador sintático através do retorno da rotina **yylex()**, como já foi detalhado antes. O analisador sintático LALR(1) então decide se está pronto para reconhecer um handle de produção (neste caso, pronto para reduzir), se deve ler mais um token do analisador léxico (shiftar), ou se deve emitir uma mensagem de erro. Tendo reduzido um handle, a ação semântica correspondente é executada, possibilitando análise semântica e geração de código dirigida pela sintáxe[2]. Tendo shiftado, o próximo token é requisitado e a decisão anterior é repetida. Tendo detectado um erro sintático, o programa desvia para a rotina **yyerror()**. Note que é possível especificar ações semântica no meio do corpo de produções (o comum é no final, para que seja executada depois de uma redução). Inclusive fazemos isso para implementar a manutenção das tabelas de símbolo por escopo; reveja a figura 7. O importante, nesses casos, é evitar a possível ambiguidade em que duas ou mais produções diferentes são consideradas num mesmo momento (ou seja, o parser está num estado

[1] O ***Bison*** gera um parser que executa ações semânticas. Este parser é principalmente utilizado para implementar esquemas de tradução dirigida pela sintaxe. Como ainda não estamos na entrega GCI, omito, por enquanto, as capacidades do ***Bison*** de produzir um gerador de código intermediário.

[2] Note que não é necessário utilizar-se do tempo de análise sintática para fazer análise semântica e/ou geração de código intermediário. Uma alternativa possível é utilizar as ações semânticas para gerar uma árvore de sintáxe representando a estrutura sintática vista. Assim, as etapas de análise semântica e geração de código intermediário podem ser feitas após a análise sintática ter sido concluida. Esta abordagem dá origem a um compilador de múltiplas passadas, em contraste ao tradicional compilador dirigido pela sintáxe.

em que não sabe qual produção irá reduzir) e pelo menos uma delas contém uma ação no meio do corpo da produção[3].

Sobre o formato do arquivo .y, ele é muito similar ao formato .l do ***flex***: possui três seções delimitadas por %%. As seções são as mesmas, exceto pela do meio, que agora contém a gramática LALR(1) anotada por ações semânticas. Na seção de definições, é possível (e fazemos isso) especificar tokens (vide diretiva %token no nosso código fonte) e o tipo C utilizado pela parser para armazenar não terminais em sua stack (vide diretiva '%nterm <ival> type' no fonte). Aqui é importante ressaltar que tokens são armazenados na stack conforme sua definição (a mesma utilizada pelo ***flex***); e.g., **INT_CONST** é armazenado como um inteiro e **IDENT** como string. Por fim, a sintaxe esperada da gramática anotada é **var1: prodbody1 < possível acao1> prodbody2 < possível _acao2> … prodbodyk < possível _acao_final>.** Ações são trechos de código **C/C++** entre '**{ }**'. A notação é bem similar à convencional utilizada por livros como o do Dragão.

São vários os exemplos do ***Bison*** em ação. Um particularmente famoso é a calculadora infixada, que pode também ser encontrada na documentação oficial do ***Bison.***

[3] Exemplificarei o problema: considere as produções **P1** e **P2**, ambas com o mesmo começo. A única diferança é que num dado pedaço desse prefixo igual, **P1** tem uma ação semântica embutida enquanto **P2** não tem. Qualquer ação que o parser tomar pode terminar num erro fatal: Se o parser executar a ação associada a produção **P1**, e depois perceber que o caminho certo era **P2**, ele não terá como desfazer os possíveis efeitos colaterais de sua ação (colocar um símbolo na tabela de simbolos, por exemplo). Analogamente, se ele não agir, provavelmente causará um erro mais a frente quando o resultado da ação será esperado (pense no erro 'variável não declarada' que poderia acontecer tendo o parser reduzido uma declaração sem adicionar o símbolo na tabela de símbolos!)

TL;DR: o parser não tem como determinar o que fazer neste caso. Assim, esse tipo de ambiguidade deve ser evitado pelo projetista usuário do ***Bison***.