CWAM-Prolog: Implementação de tradutor Prolog em C baseado na Máquina Abstrata de Warren

Renato Avellar Nobre[1]
15/0146698

Departamento de Ciência da Computação, Instituto de Ciências Exatas, Universidade de Brasília, Distrito Federal, Brasil
rekanobre@gmail.com
www.github.com/Skalwalker

Resumo Essa versão do documento apresenta uma introdução à proposta de implementação da linguagem Prolog em C[1].

Palavras-Chave: Tradutores · Máquina Abstrata de Warren · Programação em Lógica · Prolog.

1 Introdução

Linguagens de programação são notações para descrever computações para pessoas e para máquinas [1]. Para uma linguagem de programação poder ser executada, precisa primeiramente ser traduzida para alguma forma acessível para o computador. Quem realiza essa transformação de linguagens de programação, entendida por programadores, para uma linguagem entendida pela máquina, são softwares tradutores [1].

Esse artigo descreve o projeto e implementação da linguagem de programação Prolog, seguindo os conceitos da Máquina Abstrata de Warren (WAM). A linguagem foi concebida em 1970 por Alain Colmerauer na Universidade de Marseille e foi a primeira concretização do conceito de programação em lógica [2]. A ideia base do paradigma de programação dessa linguagem é que a computação pode ser expressa como uma dedução controlada de um fato declarativo. Posteriormente, uma versão refinada e abstrata dos princípios da linguagem Prolog veio a tona, que é denominada Máquina Abstrata de Warren. A WAM consiste de uma arquitetura de memória e um conjunto de instruções feitos sob medida para o Prolog, podendo ser implementado de forma eficiente em uma ampla gama de arquiteturas [2]. Sendo assim considerada o padrão base para a implementação da linguagem [2].

[1] Esse trabalho não tem como objetivo a implementação de uma nova primitiva na linguagem. Sua proposta foi discutida com a professora e no lugar de implementar a primitiva, implementarei um interpretador. A professora veem orientando os alunos sobre a correção do mesmo via email, confira se você leu as orientações enviadas por ela antes de realizar as correções.

Nesta seção serão reforçados conceitos de programação em lógica, apresentando uma visão geral de alto nível da linguagem Prolog. Serão apresentados motivos para escolha da linguagem, motivos de adequação para soluções de programação em lógica, problemas que são resolvidos, facilidades da linguagem para o programador e dificuldades para implementação de seu tradutor. Posteriormente, uma descrição formal e informal da linguagem serão apresentadas juntamente com uma breve descrição semântica.

1.1 Programação em Lógica

O conceito de paradigma de programação em lógica surgiu em 1970 como uma consequência direta do crescimento de automação de provas de teorema e programação em inteligência artificial [3]. A ideia de usar lógica de primeira ordem como linguagem de programação foi revolucionária. A consideração e linearidade em deduções lógicas proporcionou a implementação como paradigma de programação.

Uma das principais ideias do paradigma é que um algoritmo constitui-se de dois componentes disjuntos, a lógica e o controle. A lógica é a declaração de qual problema precisa ser resolvido; e o controle, como o problema será resolvido.

A motivação de programar com tal conceito se dá ao fato de que a lógica providencia um formalismo único para diversas áreas da Ciência da Computação. Aplicações de risco necessitam desse formalismo para poder analisar a corretude de seus algoritmos implementados. Portanto, aplicações desse formato junto com a simplicidade e elegância da programação lógica assegura a necessidade deste paradigma se tornar uma unidade básica da computação.

Dentro desta área, o principal nome de linguagem usado por pesquisadores é o Prolog (PROgraming in LOGic).

1.2 O Prolog

Quando falamos de programação em lógica, Prolog é o principal nome da área, o que justifica sua escolha. No entanto, a linguagem não é muito popular nem muito usada por programadores novos que não são da área de provas de teorema. Este artigo apresenta CWAM-Prolog, uma abordagem que busca a implementação do Prolog em C, utilizando os conceitos da Máquina Abstrata de Warren.

Prolog é adequada para programação em lógica porque resolve problemas de provas de teorema usando um fragmento da lógica de primeira ordem. A execução de um programa em Prolog é a construção de uma prova e utiliza conceitos fundamentais da programação como unificação, recursão e *backtracking*.

Devido à sua funcionalidade, a linguagem não provê estruturas clássicas de linguagens de programação, como estrutura de controle de fluxo ou de repetição. Para usuários leigos, a quantidade de gramática que o usuário tem que se familiarizar é relativamente pequena e sua tipagem é simples. Prolog não implementa tipos de dados da mesma forma que outras linguagens. Todo dado é considerado um termo, cuja natureza depende da forma em que foi declarado.

No entanto, sua implementação é longe de trivial, considerando que sua execução é dada por avaliações de consultas e o processo de avaliação é realizado pela resolução linear com função de seleção para cláusulas definidas (Resolução SLD). Acredita-se que o desenvolvimento do interpretador Prolog, baseado no modelo da máquina abstrata, será o maior desafio encontrado no projeto. Durante do desenvolvimento do interpretador, acredita-se que a maior dificuldade será relativa as otimizações, tais como: otimizações de última chamada, regras encadeadas, indexação e implementação da regra de corte.

1.3 Visão Geral

CWAM-Prolog funciona prioritariamente em termos de primeira ordem, onde termo é definido como uma variável, constante ou estrutura da forma $f(t_1, \ldots, t_n)$ onde f é um funtor (definido como constante) e t_i um sub-termo de primeira ordem [2]. Em outras palavras, o termo é o único tipo de dado presente, mas pode representar: átomos, números, variáveis ou termos compostos. Átomos são nomes de uso geral sem significado inerente. Números podem ser *floats* ou inteiros. Variável é como uma incógnita, cujo valor é desconhecido a princípio mas, após descoberto, não sofre mais mudanças. E por último, termos compostos são criados a partir de um funtor que é necessariamente um átomo e parâmetros listados, onde o número de parâmetros é chamado de aridade. A partir dos termos compostos, pode-se estruturar também tipos de dados de lista e *string*.

A aplicação considerada de Prolog é divida em Programas e Consultas. Um programa descreve relações definidas por meios de cláusulas. Tais cláusulas podem assumir dois tipos: fatos e regras. Fatos consistem de uma cabeça, que é um átomo e uma sequência de argumentos. Exemplificando, *humano(Chomsky)* indica que Chomsky (o argumento) é humano. Já uma regra consiste de uma cabeça que implica em um corpo, no formato: $Cabeça :- Corpo$, onde um corpo pode conter um ou mais fatos. Uma regra é lida como, "a cabeça é verdade se o corpo é verdade"

Posteriormente, a execução é realizada em cima do programa. Uma execução em Prolog é uma consulta do usuário. Essa consulta é carregada sobre uma série de processos de resoluções SLD, o que significa que se a consulta negada puder ser refutada, segue-se que a consulta é consequência lógica do programa.

1.4 Descrição Sintática

Um bom entendimento da linguagem necessita de um bom entendimento da sua forma, ou seja, da sua sintaxe. Uma gramática livre de contexto, ou simplesmente gramática, é utilizada para definir o escopo da sintaxe da linguagem. A seguinte gramática descreve de forma concisa e aproximada a sintaxe do CWAM-Prolog [4].

- programa → predicado
- predicado → cláusula | predicado cláusula
- cláusula → fato | regra

- fato → estrutura .
- regra → estrutura :- estruturas .
- estruturas → estrutura | estrutura, estruturas
- estrutura → **CON** | **CON** (argumentos)
- argumentos → termo | termo, argumentos
- termo → estrutura | lista | **VAR**
- lista → [ ] | [termo] | [termo | termo]

Um programa em Prolog é considerado uma base de dados, na qual contém um ou mais predicados e cada predicado uma ou múltiplas cláusulas. Uma cláusula pode assumir dois formatos, fato ou regra, dos quais a definição já foram discutidas previamente. Cláusulas são compostas de estruturas que são funtores seguidos de zero ou mais argumentos.

Argumentos em Prolog podem ser qualquer termo aceito pela linguagem ou variáveis. Uma variável é escrita como uma sequência de símbolos alfanuméricos que começa com uma letra maiúscula.

Finalmente, a linguagem aceita também diversos operadores infixos, listas, strings e comentários. O suporte dessas funcionalidades serão adicionados de forma gradual na gramática e no projeto. No entanto, note que operadores infixos podem ser considerados estruturas onde o operador é o nome e os operandos os seus argumentos.

1.5 Descrição Semântica

A implementação CWAM-Prolog considera um interpretador de Prolog baseado em C. A execução do programa de dados será dada por *cwamp arquivo.pl* e um terminal para digitar os comandos de consulta será executado sobre o arquivo desejado. Um arquivo com extensão *.pl* apresentará a seguinte forma:

```
1    ama(edward, bella).
2    ama(jacob, bella).
3    ama(coelho,cenoura).
4
5    ciumes(X,Y) :- ama(X,Z), ama(Y,Z).
```

Observe que esse arquivo apresenta fatos e uma regra. Posteriormente uma consulta, após a execução do *cwamp*, pode ser realizada da seguinte forma.

```
1    ?- ciumes(edward,W).
```

A linguagem CWAM-Prolog leva em consideração variáveis sendo sempre como letra maiúsculas ou o símbolo *underscore* e fatos como letra minúscula. Depois que um valor é escolhido para uma variável, ele não pode ser alterado pelo código subsequente; entretanto, se o restante da cláusula não puder ser satisfeito, Prolog pode retroceder e tentar outro valor para aquela variável. O escopo de um nome de variável é a cláusula em que ocorre.

Sendo assim, precisa-se definir a semântica das cláusulas. A semântica declarativa das cláusulas definidas nos diz quais objetivos podem ser considerados

verdadeiros de acordo com um determinado programa e são definidos recursivamente da seguinte forma: Um objetivo é verdadeiro se for a cabeça de alguma cláusula e cada um dos objetivos do corpo dessa cláusula for verdadeiro, onde uma cláusula (ou termo) é obtida substituindo suas variáveis, por um novo termo para todas as ocorrências da variável [6].

A semântica da execução define exatamente como o sistema Prolog executará um objetivo e as informações de sequenciamento são os meios pelos quais o programador direciona o sistema para executar o programa. O resultado da execução de um objetivo é enumerar todas as possíveis instâncias verdadeiras [6].

Outras informações importantes:

- **Inteiros** Apenas em base 10, podendo aceitar números positivos e negativos.
- **Floats** Positivos e negativos, contendo um ponto para separar a parte fracional. Não são aceitos números em formatos científicos.

2 Analisador Léxico

Nesta seção demonstraremos como o analisador léxico do CWAM-Prolog foi implementado. Um analisador léxico é responsável pelas fases iniciais da compilação, sendo sua principal tarefa ler os caracteres de entrada do arquivo fonte, identificar lexemas e produzir como saída uma sequência de *tokens* para cada lexema do programa fonte [1]. A sequência de *tokens* gerados a partir do lexema vai ser posteriormente usada pelo analisador sintático.

O analisador léxico pode ser produzido de forma automática, ao especificarmos os padrões para um gerador de análise léxica e compilarmos esses padrões em um código que funciona como um analisador léxico [1]. Essa abordagem de implementação facilita a modularização e reutilização do analisador léxico, visto que se desejarmos fazer uma modificação, precisaremos reescrever apenas os padrões afetados e não o programa inteiro [1]. Para a implementação no CWAM-Prolog, escolheu-se um gerador de análise léxica denominado FLEX. Sua escolha providenciará uma maior agilidade na implementação do analisador, visto que será necessário apenas mexer em um programa de alto-nível para escrever os padrões [1].

É comum também que o analisador léxico interaja com a tabela de símbolos da linguagem sendo implementada. Quando o analisador descobre um lexema constituindo de um identificador, ele precisa adicionar esse lexema a tabela de símbolos do tradutor da linguagem [1].

Outro conjunto de tarefas de responsabilidade do analisador é a realização de tarefas que dependem da leitura do código fonte. Algumas que podem ser implementadas são: exclusão de comentários e espaço em brancos, correlacionar mensagens de erro do compilador ao código fonte e até mesmo expansão de macros em linguagens que utilizam tais processadores.

2.1 Gramática da Linguagem

O analisador léxico se desenvolve de acordo com a gramática da linguagem. Seguiu-se como base para o desenvolvimento do projeto a gramática, já apresentada na Subseção 1.4. Nenhuma alteração foi realizada nessa etapa.

No entanto, seus símbolos terminais, como números, átomos e variáveis, são descritos mais a frente como expressões regulares. Tais expressões são essenciais para o gerador de análise léxica ser desenvolvido.

Tokens e Tabela de Símbolos

Com a gramática definida podemos ter uma noção dos *tokens* que o analisador léxico irá encontrar. Esses *tokens* podem ou não conter lexemas. Quando os contém, estes são armazenados na tabela de símbolos do nosso tradutor [1]. A Tabela 1 descreve os quatro tipos de símbolos encontrados em implementações de Prolog puro. Qualquer tipo de dado pode ser guardado em um desses quatro *tokens*.

Tokens	Descrição Informal	Exemplo de Lexema
REF	Endereço de armazenamento de uma variável	X, Y, Z, _
STR	Endereço de armazenamento de um funtor f/n	$f(t_1, \ldots t_n)$
LIS	Endereço de armazenamento do primeiro elemento da lista	$[Z, W]$, []
CON	Valor da constante encontrada	a, b, c

d

Tabela 1. Tabela de símbolos com *tokens* e seus possíveis Lexemas encontrados em programas Prolog.

Tirando os *tokens* que apresentam lexemas, existem também aqueles que só possuem seus identificadores. A Tabela 2 mostra quais são encontrados na linguagem e devem ser tratados pelo analisador criado. Portanto, com os *tokens* e lexemas definidos, pode-se começar a análise léxica utilizando uma ferramenta que permite a especificação de suas regras de análise com base em expressões regulares, o FLEX.

<:- >	<(>	</* >	<[>	<| >	<% >
<. >	<) >	<*/ >	<] >	<, >	

Tabela 2. Tabela de *tokens* sem existência de lexemas.

2.2 Implementação do Analisador com FLEX

O FLEX é uma ferramenta que permite a especificação de um analisador léxico através de expressões regulares para descrever padrões. Sua função é transformar

os padrões de entrada em um autômato finito determinístico e gerar um código, de extensão *yy.c*, que simula tal autômato [1].

Um programa em FLEX funciona com a definição de declarações, regras de traduções e funções auxiliares [1]. As declarações e regras de tradução são os pontos principais do programa FLEX desenvolvido a serem abordados nesse trabalho. As declarações dão nome para as expressões regulares criadas, enquanto as regras de tradução convertem a expressão regular para um *token* esperado [5]. As declarações e traduções foram implementadas para o CWAM-Prolog e se encontram no arquivo *CWAMProlog/src/lexical/prolog_lex.l*, e seguem os seguintes padrões:

- Inteiros: Sequência de dígitos entre 0 e 9, podendo ou não ser precedido do sinal negativo;
- Ponto Flutuante: Sequência de dígitos entre 0 e 9 seguido de um ponto e uma sequência de dígitos subsequentes, podendo ou não ser precedido do sinal negativo;
- Constante: Cadeia de caracteres iniciada com letra minuscula e podendo conter um travessão e uma cadeia subsequente;
- Variável: Cadeia de caracteres iniciada com letra maiúscula ou travessão, e podendo conter um travessão com uma cadeia subsequente.

Política de Tratamento de Erro

O analisador léxico tem dificuldades, sem a ajuda de outros componentes, de encontrar um erro no código fonte. Ele não é capaz de sozinho encontrar erros ortográficos, nomes de funções não declarados, erros estruturais em condicionais e repetições, dentre outros diversos tipos de erro. No entanto, suponha uma situação em que o analisador léxico é impedido de prosseguir porque nenhum de seus padrões definidos encontram *tokens* respectivos. Erros desse formato conseguem ser descobertos e o programador pode aplicar políticas de recuperação para o analisador continuar sua execução.

Das diversas formas de tratamento de erro que podem ser aplicadas para um analisador léxico, escolheu-se o "modo pânico" para ser implementado. Esse formato consiste em apagar caracteres sucessivos da entrada até o analisador léxico encontrar um token definido no começo da entrada restante [1]. Apesar desse modo de recuperação ser uma opção que pode muitas vezes gerar confusões no analisador sintático, sua escolha pode ser adequada para ambientes de computação interativa. O interpretador CWAM-Prolog pode ser considerado um desses tipos de ambiente. A regra implementada para esse tratamento de erro é bem simples, sendo sua única requisição ser a última regra do programa FLEX, para assim possuir a menor prioridade:

- . {printf("ERROR: %s:%d:%d: Lexical error: Unrecognized Token Found: %*s*"}

3 Analisador Sintático

Após a implementação da análise léxica, é necessário realizar alguma forma de tratamento sobre os *tokens* encontrados. Essa tarefa é deixada a cargo do analisador sintático. Tal analisador, ao receber um *token* encontrado pelo analisador léxico, tem a tarefa de verificar se a cadeia de *tokens* recebida faz parte da gramática da linguagem [1]. Ademais, o analisador sintático também é responsável, em programas de entrada bem definidos, por construir uma árvore sintática e a fornecer para o resto do código para processamento futuro [1].

O projeto da maioria das linguagens de programação aproveita de regras precisas que descrevem a estrutura sintática de programas bem definidos [1]. A sintaxe de um programa é definida por construtos especificados por gramáticas livre-de-contexto em notação na forma Backus-Naur (BNF) [1]. Essa gramática para a linguagem Prolog, foi especificada na Subseção 1.4. Pequenas alterações são realizadas na gramática para o analisador sintático. A regra, *programa* $\rightarrow$ *programa predicado* foi removida por causar conflitos de *shift/reduce* e não ser necessária ao nível de Prolog que será desenvolvido. A regra *nome*, foi trocada pela adição do terminal CON. Por último uma regra de sintaxe de lista foi implementada. As regras gramaticais abaixo representam o conjunto de regras implementadas pelo analisador sintático.

Para o desenvolvimento do analisador sintático será considerado um gerador de analisadores sintáticos, denominado *Bison*. Tal gerador é capaz de gerar analisadores sintáticos LALR(1), GLR, e LR utilizando a linguagem C [1]. Nesta seção será também integrado o analisador léxico junto ao sintático. Portanto, *tokens* reconhecidos pelo analisador léxico serão enviados para o sintático, que proverá como saída a árvore sintática abstrata e a tabela de símbolos. Caso exista algum erro no arquivo de entrada, o analisador reportará o erro detalhado, incluindo sua localização.

3.1 Tabela de Símbolos e Árvore Sintática

O analisador sintático produzido deverá apresentar tanto a árvore sintática quanto a tabela de símbolos em sua saída. Essa seção se preocupa em descrever como a tabela de símbolos será implementada, bem como qual deve ser a estrutura da árvore criada.

Informações relevantes ao programa devem ser armazenadas na tabela de símbolos, tanto pelo analisador léxico quanto pelo analisador sintático, sendo cada analisador responsável por adicionar uma parte da informação na tabela.

No entanto, a Máquina Abstrata de Warren, por padrão, não implementa uma tabela de símbolos [2]. Deve se pensar na WAM como uma espécie de código de máquina, não existem tabelas de símbolos nesse tipo de código [2]. No entanto, pode haver uma seção separada da WAM no executável do CWAM-Prolog, que prove informações que outras ferramentas possam usar. Portanto,

tabelas de símbolos temporárias serão criadas durante a análise sintática, não sendo necessária sua consulta e atualização durante a execução.[2]

Durante a execução o CWAM-Prolog utiliza uma *heap* que guarda o estado de sua execução. Como Prolog manipulará essa *heap* durante sua interpretação, seus tipos são guardados diretamente nela [2]. No analisador sintático, uma tabela de símbolos será criada apenas com o propósito de encontrar possíveis erros léxicos ou sintáticos na linguagem e auxiliar na criação da *heap*. A tabela deverá conter, tipo do *token* (i.e. REF, STR, CON), e seu lexema respectivo, que será utilizado como chave identificadora na tabela. Observe que o tipo LIS não precisa ser adicionado na tabela de símbolos porque todas as listas Prolog são anônimas e dizem respeito somente a regra ou fato em que estão escritas. A tabela de símbolos foi implementada utilizando uma tabela *hash*, disponível em uma biblioteca C denominada *uthash*.

Finalmente, o programa deverá mostrar em sua saída além da tabela de símbolos, a árvore sintática de sua execução. Tal árvore será construída individualmente para cada regra ou fato, utilizando de estruturas para contribuir em sua criação.

Para auxiliar na implementação das funcionalidades algumas funções foram escritas e padrões estabelecidos. Existe uma estrutura genérica para todos os tipos importantes da gramática. Essa estrutura foi criada com uma *union* que decide o tipo de dado a ser utilizado e também tem sua função específica para criação, exibição e para liberação da memória. Adicionalmente a função *yyerror* foi alterada para mostrar as linhas e colunas de onde ocorreu o erro.

3.2 Política de Tratamento de Erros

Uma tarefa crucial do analisador sintático é implementar funcionalidades que ajudem o programador a localizar e remover erros em seu código [1]. A maioria das linguagens de programação, Prolog incluso, não especificam como os erros devem ser tratados, deixando essa tarefa a cargo do arquiteto do compilador [1]. Planejar a política de tratamento de erros com antecedência pode ser capaz de simplificar a estrutura do projeto, além de melhorar a detecção de erros [1]. Os seguintes erros comuns são considerados até o momento:

- **Erros Léxicos:** Erro ortográfico de identificadores, palavras chaves ou operandos, ou aspas faltantes em strings;
- **Erros Sintáticos:** Ponto e vírgula mal posicionados, parênteses, chaves ou colchetes adicionais/faltantes, dentre outros.

A precisão dos analisadores sintáticos permite que tais erros sejam encontrados de forma bem eficiente [2]. Métodos LL e LR de análise detectam os erros o mais rápido possível, ou seja, quando a cadeia de caracteres não consegue ser analisada mais adiante de acordo com a gramática da linguagem [2]. Portanto, a

[2] Vale ressaltar que as variáveis no Prolog são variáveis lógicas. Seus valores depois de descobertos não são alterados.

política de identificação e tratamento de erro deve se adequar as seguintes premissas: (i) relatar a presença de erros de forma clara e correta; (ii) se recuperar de cada erro de forma rápida o suficiente para detectar erros subsequentes; (iii) adicionar o mínimo de custo de processamento para programas funcionais.

A implementação da política de identificação de erros adotada nesse trabalho apresentará a linha e a coluna em que o erro ocorreu. Além de mostrar também o símbolo equivocado e o tipo de símbolo que se espera. No entanto, após o programa detectar um erro, ele ainda precisa arranjar alguma forma de se recuperar. Apesar de nenhuma estratégia ter se provado amplamente aceita, alguns métodos são comumente utilizados. Dentre os mais comuns está o método de recuperação em modo pânico. O mesmo foi escolhido para ser implementado nesse trabalho.

A implementação do método de recuperação em modo pânico consiste em descartar símbolos de entrada, um por um, até que um *token* de um conjunto de *tokens* de sincronização seja encontrado [1]. Esse conjunto de *tokens* é escolhido pelo projetista de acordo com a linguagem sendo implementada. Para o projeto CWAM-Prolog, o *token* de sincronização utilizado é o ponto final das expressões.

A escolha do modo pânico, pode acarretar perda de alguns erros sintáticos, devido a sua forma de eliminar os símbolos [1]. No entanto, tem como vantagem a simplicidade e garantia que não vai ficar preso em uma estrutura de repetição infinita [1].

3.3 Implementação do Analisador Sintático

Com a estrutura da tabela de símbolos, especificação da árvore sintática e a política de tratamento de erros definida, pode-se inicializar a descrição de como a implementação do analisador foi realizada.

Recapitulando, o analisador sintático foi implementado utilizando o gerador Bison. O Bison, por padrão, implementa um algoritmo LALR(1), que aceita a linguagem definida pela gramática livre de contexto do CWAM-Prolog, descrita na Seção 2.1. Um programa Bison tem como as seções prólogo, declarações, regras sintáticas e epílogo. Sua seção mais importante é a das regras sintáticas. Ela é responsável por receber um *token* do FLEX e encaixá-lo na gramática livre de contexto proposta. As regras sintáticas no código do Bison possuem o seguinte formato:

```
1   <variavel>: <corpo> ;
```

A variável, é o símbolo não terminal definido por essa regra. O corpo são sequências de terminais e não terminais que compõem esta regra, separadas pelo símbolo "|". Portanto, no CWAM-Prolog, cada regra da gramática revista é transformada em uma regra sintática no formato do Bison. Nessas regras implementadas, ocorrem chamadas para criação de nós, adição na tabela de símbolos (quando necessário), visualização e liberação de memória. As regras sintáticas, em formato aceito pelo Bison, estão definidas no arquivo: *CWAMProlog/src/-parser/prolog_bison.l*.

4 Analisador Semântico

Na maioria das linguagens de programação o compilador/interpretador tem como tarefa, além de gerar código executável, através de análises léxicas e sintáticas, verificar se o código fonte segue as regras semânticas definidas pela linguagem [1]. A análise semântica faz parte da análise estática do código, na qual não é necessária sua execução para a detecção de erros [1]. A verificação estática é capaz de descobrir diversos tipos de erro de programação e reportá-los durante a compilação. Dentro da análise estática existem duas categorias principais de análise:

- Checagem Sintática: Esse tipo de checagem não ocorre somente pelo analisador sintático, há mais na sintaxe do que simplesmente a gramática [1]. Por exemplo, garantir que identificadores sejam definidos somente uma vez no mesmo escopo ou que um comando *break* esteja dentro de uma estrutura de repetição;
- Checagem de Tipos: As regras de tipo de uma linguagem asseguram que um operador ou uma função seja aplicado a uma quantidade e tipo certos de operandos. Cuidando também de realizar conversões implícitas de tipo quando necessárias, como somar um inteiro e um decimal [1].

A checagem estática geralmente é feita durante a construção da árvore sintática para o código fonte. No entanto, análises semânticas mais complexas precisam ser feitas em duas etapas, primeiro construindo uma representação intermediária e depois a analisando [1].

A análise estática é possível graças a técnica de tradução direcionada-a-sintaxe. Tal técnica, utiliza um mecanismo de associar atributos aos símbolos da gramática. Uma definição direcionada-a-sintaxe especifica os valores dos atributos, associando regras semânticas às produções gramaticais.

Nesta seção será descrito como a análise semântica foi implementada no CWAM-Prolog, utilizando regras gramaticais livres de contexto associadas com fragmentos de programas em suas produções. Essa abordagem é denominada esquema de tradução direcionada-a-sintaxe (SDT). Para implementar essa abordagem, não foram necessárias alterações na estrutura da gramática da linguagem. No entanto, algumas mudanças léxicas foram realizadas para tratar erros semânticos do Prolog. Todo o seu processo de implementação será descrito detalhadamente na seção seguinte.

4.1 Implementação do Analisador Semântico

Diferentemente das análises léxicas e sintáticas, a análise semântica não utilizou um gerador de analisador. Para sua implementação utilizou-se técnicas baseada em anotação da gramática e esquemas de tradução disponíveis no Bison. A implementação do analisador foi feita em uma fase somente, sem realizar a etapa de representação intermediária, pois a semântica do Prolog é simples o suficiente para ser feita dessa forma.

Semântica Implementada

Diferente da maioria das linguagens convencionais, o Prolog puro não possui muita semântica. O Prolog não possui regras de escopo de variáveis, não contém tipos de dados diferentes e não possui funções. Portanto, não precisando se preocupar com regras de passagem de parâmetros nem conversões de tipos.

No entanto, compiladores Prolog ainda apresentam algumas mensagens de avisos realizados pela análise semântica. Os tipos de aviso apresentados a seguir serão os implementados no CWAM-Prolog:

- **Descontinuidade:** Predicados com o mesmo nome e mesma aridade devem vir seguidos no código. Apesar de criar somente um aviso na compilação, em geral, é um erro de programação e o seu usuário deve ser avisado. Um exemplo de um caso que geraria esse aviso é o seguinte:

```
1        dono(renato).
2        cachorro(jony).
3        dono(giovanna).
4        cachorro(jojo).
```

  A forma correta deveria ser:

```
1        dono(renato).
2        dono(giovanna).
3        cachorro(jony).
4        cachorro(jojo).
```

- ***Singleton:*** Variáveis que aparecem somente uma vez na mesma cláusula, geralmente são erros de programação e o programador deve ser, ao menos, avisado. No entanto, nem todas às vezes que isso acontece é um erro de programação. Para tratar esses casos, o programador deve adicionar o símbolo "_" antes da variável, e o analisador irá suprimir o erro. Os seguintes casos podem ocorrer ao analisar *singletons*:

```
1        % Aviso variavel singleton: X
2        teste(X, renato).
3        % Sem erros
4        teste(_X, renato).
5        % Aviso variavel marcada como singleton
6        % aparecendo multiplas vezes
7        teste(_X, _X).
8        % Sem erros
9        teste(_, _).
10       % Sem erros
11       teste(_).
```

  Observe nos dois últimos casos, que uma variável anônima representada somente por "_" não entra nas regras para *singleton*.

Ainda existem outros casos de semântica relativa à linguagem Prolog. No entanto, tais casos serão implementados somente na máquina abstrata, e são descritos aqui somente para aumentar a compreensão em relação a linguagem:

– **Regras de Escopo:** As variáveis no Prolog são lógicas e, portanto, não declaradas. Sendo assim, o escopo é definido como o corpo da cláusula onde a variável é usada. Na tabela de símbolos, implementou-se na chave o escopo como sendo a linha em que aparece o predicado.
– **Passagem de Parâmetros:** O Prolog define como regra que a passagem de parâmetros seja feita sempre por cópia

Quanto à semântica que foi implementada, a análise de descontinuidade e *singletons* foi realizada com o auxílio de algumas funções e estruturas adicionais. Foram implementadas duas tabelas *hash* adicionais, para incluir, contar, e manter informações de posição das variáveis e dos predicados. Adicionalmente, algumas estruturas nos analisadores léxico e sintático foram modificadas.

Alterações nos Analisadores Léxico e Sintático

Em relação ao analisador léxico, a identificação da variável foi alterada. Para facilitar a análise semântica, o analisador léxico identifica três tipos distintos de variáveis, variável anônima ("_"), variável *singleton* marcada (variáveis começando com "_") e as variáveis normais. Para cada tipo de variável adicional é criado um *token*, e a identificação desse *token* é adicionada na gramática.

Posteriormente, uma alteração importante foi realizada na tabela de símbolos. Agora estruturas precisam ser adicionadas na tabela de símbolos considerando sua aridade, pois o Prolog considera estruturas com mesmo nome, mas quantidade de símbolos diferentes, estruturas diferentes. Portanto, um funtor *estuda(x, y, z)* será adicionado na tabela de símbolos como *estuda/3* e um funtor *estuda(x, y)* será adicionado como *estuda/2*.

Árvore Sintática Anotada

Com a análise semântica implementada, pode-se criar uma árvore sintática anotada com as informações de auxílio aos tratamentos. No CWAM-Prolog a árvore sintática anotada adicionou novas informações referentes ao tipo para seus nós mais importantes. Os termos da gramática aparecem agora seu tipo específico e, caso seja uma variável, o tipo de variável:

```
1    <termo> X (tipo: REF; tipo_var: VAR)
2    <termo> _X (tipo: REF; tipo_var: SINGLETON VAR)
3    <termo> _ (tipo: REF; tipo_var: ANONYMOUS VAR)
```

Estruturas com argumentos apresentam sua aridade, e estruturas sem argumentos aparecem como tipo constante:

```
1    <estrutura> estuda (aridade: 2)
2    <estrutura> csc134 (type: CON)
```

Fatos e regras apresentam a posição em que aparecem no código fonte:

```
1    <regra> (pos: 22)
2    <fato> (pos: 18)
```

Tendo em consideração à tabela de símbolos, além das mudanças realizadas descritas anteriormente, diversos outros dados foram adicionados. A tabela de símbolos passou a indexar os elementos com informações do nome, aridade (quando existente) e escopo. No Prolog o escopo é único para cada linha de código. Sendo assim, foi utilizado a posição no código fonte para defini-lo. Foram adicionadas também informações sobre aridade, tipo de variável e posição no código fonte.

4.2 Política de Tratamento de Erros

Tendo a análise semântica implementada, todos os tipos de análise do código fonte foram realizados. O tradutor implementado é capaz de identificar, e se recuperar, de erros léxicos, sintáticos e semânticos.

Durante o desenvolvimento do trabalho, escolheu-se as políticas de erro para cada uma das fases. No analisador léxico a política escolhida foi a de modo pânico, que consiste em apagar caracteres sucessivos da entrada até o analisador léxico encontrar um token definido no começo da entrada restante. Sua escolha foi devida a sua adequação em ambientes de computação interativa, que é o caso do CWAM-Prolog. No analisador sintático, também foi implementada uma política de modo pânico. Essa politica foi escolhida por não ter a possibilidade do compilador ficar preso em uma estrutura de repetição infinita. Por fim, no analisador semântico, nenhuma política de tratamento de erros precisa ser implementada para o CWAM-Prolog. Por ter sua semântica simplificada, em nenhum momento durante sua tradução o tradutor irá encontrar um erro em que não consiga se recuperar.

Por fim, o tradutor CWAM-Prolog se preocupa em ser detalhado quanto aos seus erros, mostrando nome do arquivo, linha e coluna, tipo do erro (léxico, sintático ou semântico) e descrição do erro ocorrido. Sendo assim, o programador tem todo o suporte necessário para consertar seu código fonte.

5 Implementando a Máquina Abstrata de Warren

Com os analisadores léxico, sintático e semântico implementados, pode-se passar para a etapa final da implementação do CWAM-Prolog. Essa etapa consiste em construir um interpretador para a linguagem. Esse interpretador será implementado através de uma máquina abstrata, seguindo o modelo de Warren, David HD (1983) [7]. A Máquina Abstrata de Warren (WAM) foi projetada contendo uma estrutura de memória e um conjunto de instruções e pode ser considerada o padrão *de facto* para implementação de compiladores *Prolog*. A Figura 5, demonstra a arquitetura de memória implementada, e seus registradores. As instruções implementadas serão descritas posteriormente.

Nesse trabalho a implementação da WAM foi desenvolvida segundo o tutorial de Ait-Kaci, Hassan (1991) [2], que simplifica o texto de Warren, David HD (1983) [7], para fácil implementação. A implementação do CWAM-Prolog é feita então de forma construtiva, adicionando complexidade às linguagens e

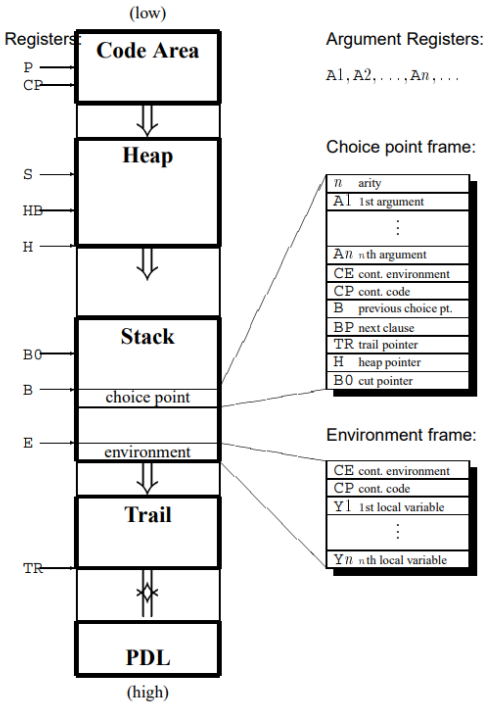

Figura 1. Arquitetura de Memória da WAM [7]

suas respectivas máquinas em diversas etapas. Desta forma de projeto, conseguimos isolar e testar diversos aspectos do Prolog. Sendo assim a Subseção 5.1 vai expandir sobre o o processo de unificação, implementando linguagens L_0 e L_1 com máquinas M_0 e M_1. Posteriormente, na Subseção 5.2 implementaremos a resolução plana com a máquina M_2, que funcionará igual ao Prolog, mas sem a possibilidade de *backtracking*. Seguindo pela Subseção 5.3 em que o *backtracking* é implementado, construindo uma M_3 que funciona para programas Prolog puros. Finaliza-se então, na Subseção 5.4 onde algumas otimizações ao projeto são implementadas.

5.1 Unificação

Começaremos primeiro implementando a unificação. Lembre-se que um termo em Prolog pode ser uma variável, uma constante ou uma função (onde f/n é a função f com aridade n) [2]. Para implementar a unificação, definimos uma linguagem L_0, que pode conter dois tipos de entidades, um termo de programa e um termo de pesquisa. Um interpretador para L_0 pode ser definido como uma máquina abstrata M_0, constituída de uma representação dos dados disponíveis e de um conjunto de instruções, que utilizará um algoritmo de unificação como semântica

operacional [2]. Portanto, a M_0 avaliará como sucesso ou falha a unificação entre o programa e a pesquisa [2].

Representação dos Termos

Na linguagem L_0, os termos serão armazenados em uma *heap*, como apresentada na Figura 5.1. Essa *heap* armazenará dois tipos de dados, variáveis e estruturas. Uma variável é identificada como um ponto de referência e representada utilizando uma única célula na *heap*, identificada como $< REF, \# >$ onde $\#$ é a posição de endereçamento. Por outro lado, uma estrutura (qualquer termo não variável, incluindo constantes), vai ser apresentada na *heap* por uma tag f/n seguida de um valor $< STR, \# >$.

0	STR	1
1	$h/2$	
2	REF	2
3	REF	3
4	STR	5
5	$f/1$	
6	REF	3
7	STR	8
8	$p/3$	
9	REF	2
10	STR	1
11	STR	5

Figura 2. Representação da *heap* para $p(Z, h(Z, W), f(W))$. [7]

Compilando Pesquisas

Realizar a unificação, na semântica operacional da L_0, implica em traduzir uma pesquisa em uma sequência de instruções para construí-la na *heap* [2]. Para auxiliar na construção da *heap*, a máquina M_0 conta com um número de registradores de variáveis $X1, x2, \ldots, X_n$. Esses registradores, serão considerados temporários e guardarão uma célula de *heap*, contendo o termo, durante o processo de construção. Os registradores serão alocados, utilizando sempre o menor índice disponível, seguindo a ordem dos termos mais externos para os termos internos [2]. Por consequência, variáveis externas serão esquecidas e uma pesquisa pode ser transformada para sua forma nivelada. Posteriormente, uma geração de código da esquerda para a direita vai ser realizada sobre a a forma nivelada,

gerando uma sequência de tokens[3] [2]. Para implementar todo esse processo a M_0 utiliza as seguintes instruções:

- **put_structure** $f/n\ Xi$: Quando encontrado um registrador com uma estrutura, coloca essa estrutura na *heap* e copia o endereço para o registrador;
- **set_variable** Xi: Quando encontrado um registrador não encontrado antes, coloca um novo REF na *heap* contendo seu próprio endereço, e o copia para o registrador;
- **set_value** Xi: Quando encontrado um registrador conhecido, coloca a nova célula na *heap* e a copia para o valor do registrador.

Compilando Programas

Compilar um programa, assume a existência de uma pesquisa já construída na *heap*. Portanto, unificar o programa com a pesquisa pode ser realizado seguindo o termo construído em X1, desde que mapeie funtor a funtor a estrutura do programa [2]. No entanto, existe uma complicação com novos termos presentes no programa. Portanto, um programa em linguagem L_0 precisa funcionar em dois modos, leitura e escrita [2]. Adicionalmente, um novo registrador S é implementado para guardar a posição do subtermo que precisa ser mapeado.

Quando o programa é construído na *heap*, por assumir-se que já existe uma pesquisa construída na *heap*, um termo sera criado de baixo para cima, primeiro tratando estruturas externas e por último as mais internas. As seguintes instruções são implementadas para sua construção:

- **get_structure** $f/n\ Xi$: Executada quando encontrado um registrador com uma estrutura. Caso encontre uma REf solta, a instrução mapeia para uma nova célula de estrutura f/n, a coloca na *heap*, e configura o modo para escrita. Caso encontre uma estrutura apontando para um funtor f/n, guarda seu endereço no registrador S e configura o modo para leitura.
- **unify_variable** Xi: Executada quando encontrado um registrador não encontrado antes. Em modo de leitura, armazena o valor da *heap* no endereço do registrador S no registrador. Em modo de escrita, funciona igual a set_variable.
- **unify_value** Xi: Executada quando encontrado um registrador conhecido. Em modo de leitura tenta unificar o registrador Xi com a célula da posição do registrador S. Em modo de escrita, funciona igual a set_value.

Máquina M_1

Com a unificação implementada[4], pode-se estender a linguagem L_0 em uma linguagem L_1 similar, na qual um programa pode ser um conjunto de átomos definindo no máximo um fato por nome de predicado [2]. Nessa linguagem separa-se

[3] A sequência de tokens gerada pela Figura 5.1 é: $X3, X2, X5, X4, X5, X2, X3, X4$.

[4] A implementação do CWAM-Prolog acaba na unificação. Desta seção em diante serve apenas de referência à implementação completa para trabalhos futuros.

a definição de átomos (funtores predicativos) e termos (argumentos dos predicados). Portanto, no contexto de tal programa, a execução de uma pergunta conecta à definição correta a ser usada para resolver o problema de unificação [2].

O conjunto de instruções da L_0 é estendido, e uma área na memória é reservada para guardar a área de código. Adicionalmente um registrador P é adicionado para sempre conter o endereço da próxima instrução a ser executada. A implementação desse registrador se justifica devido a existência de um conjunto de instruções de controle, que podem executar instruções de forma não-linear [2]. Na máquina M_1 as seguintes instruções de controle são consideradas:

- **call** p/n: Armazena no registrador P, o endereço da instrução com o nome p/n.
- **proceed**: Indica o final de uma sequência de instruções de um fato.

Eliminando símbolos de predicado da *heap*, o problema de unificação agora envolve fatos e uma pesquisa e, portanto, precisa resolver múltiplas equações [2]. Por convenção, uma nova nomenclatura de registrador Ai é introduzido para indicar que o registrador está sendo utilizado para armazenar um argumento de um átomo. Agora com variáveis podendo ser argumentos, situações de inserção nos registradores e na *heap* para fatos e perguntas precisam ser tratadas [2]. As seguintes instruções são adicionadas na linguagem L_1:

- **put_variable** X_n A_i: Quando encontrado uma primeira ocorrência de uma variável na i-ésima posição de uma pesquisa, coloca uma célula REF não inicializada na *heap*, e copia seu valor para os registradores Xn e Ai.
- **put_value** X_n A_i: Quando encontrado uma futura ocorrência de uma variável em uma pesquisa, copia seu valor para um registrador A_i.
- **get_variable** X_n A_i: Quando encontrada uma primeira ocorrência de uma variável na i-ésima posição de um fato, guarda seu valor em um registrador A_i.
- **get_value** X_n A_i: Quando encontrada uma futura ocorrência de uma variável em um fato, à unifica com o valor de A_i.

5.2 Resolução Plana

Agora estende-se a linguagem L_1 em uma linguagem L_2, onde procedimentos não são mais somente fatos, podendo também ser regras [2]. Cada regra possuirá um corpo definido por uma sequência conjuntiva de átomos. A semântica operacional da L_2 constitui de executar uma pesquisa e repetir a aplicação de resoluções mais à esquerda até que a pesquisa fique vazia ou um erro seja obtido [2]. As instruções criadas para a M_1 são mantidas nessa etapa, e serão usadas para as regras da L_2.

Para a implementação de regras, precisa-se guardar a informação de variáveis que são temporárias ou permanentes. Essa informação é armazenada em um frame de ambiente (veja novamente a Figura 5), que guarda informações das

variaveis Y_i locais, do ambiente de continuidade CE, e do ponto de continuidade CP [2]. Adicionalmente, um registrador extra E é criado para guardar o endereço do último ambiente no topo da pilha de execução [2]. Portanto, para implementar a pilha de execução e os ambientes, precisa-se criar duas regras adicionais de adição e remoção da pilha:

- **allocate:** Empilha um novo ambiente na pilha de execução, configurando o campo CE para o valor do registrador E, e seu ponto de continuidade para o valor de CP.
- **deallocate:** Remove o ambiente da pilha na localização do registrador E, reconfigura o valor do registrador P para o valor contido no campo CP e o valor do registrador E para o valor do campo CE.

5.3 Prolog Puro

Finalmente desenvolve-se a L_3, correspondente ao Prolog puro. A L_3 estende a L_2 ao adicionar a capacidade de realização de *backtracking*. A semântica de L_3 opera utilizando uma resolução SLD mais a esquerda de cima para baixo [2]. Portanto, uma falha de unificação na nova linguagem não gerará mais uma exceção e irá considerar outras alternativas de cláusulas, graças ao *backtracking* cronológico [2].

O projeto da M_2 será alterado para incluir um estado de computação pra cada chamada de precedimento, esse estado é denominado estado de ponto de escolha. Cada ponto de escolha será ordenado na pilha de execução. Para definições com mais de uma alternativa, um ponto de escolha será criado para a primeira alternativa, e posteriormente é atualizado por alternativas intermediárias, até ser descartado pela última alternativa. Um ponto de escolha armazenará os seguintes atributos: registradores de argumentos, ambiente atual E, ponteiro de continuação CP, registrador de ponto de escolha B, a próxima cláusula a ser executada, e os registradores TR e H [2].

Para tanto, novas instruções são adicionadas ao conjunto de instruções M_2, criando a M_3 respectiva ao Prolog puro [2]. Essas instruções são respectivas ao ponto de controle de execução e são descritas a seguir:

- **try_me_else L:** Aloca um novo ponto de escolha na pilha de execução, configurando seu campo de próxima cláusula para o valor de L, os outros campos de acordo com o contexto atual, e configura o registrador B para apontar para esse ponto.
- **retry_me_else L:** Tendo realizado *backtrack* para o ponto de escolha atual indicado no registrador B, reconfigura todas as informações necessárias e atualiza seu campo de próxima cláusula para o valor de L.
- **trust_me:** Tendo realizado *backtrack* para o ponto de escolha atual, reconfigura todas as informações necessárias e descarta o poto de escolha, reconfigurando o registrador B para seu antecessor.

5.4 Otimizações

Diversas otimizações podem ser feitas na implementação da WAM. Nesse trabalho considerou-se apenas otimizar constantes, listas e variáveis anonimas, adicionando instruções especiais. Essas otimizações salvarão espaço na *heap* na área de código e na movimentação dos dados [2]. Para tal, implementou-se as seguintes instruções, seus funcionamentos são similares as instruções já descritas e portanto não serão mais elaboradas:

- **put_constant** c, Xi
- **get_constant** c, Xi
- **set_constant** c
- **unify_constant** c
- **put_list** Xi
- **get_list** Xi
- **set_void** n
- **unify_void** n

Com isso termina-se a implementação da Máquina Abstrata de Warren e o CWAM-Prolog está pronto para ser executado e testado.

6 Testando o CWAM-Prolog

Para instalar e testar a implementação do CWAM-Prolog, são necessárias algumas dependências. É obrigatório que o ambiente de execução tenha instalado os programas flex, Bison, make, e o compilador gcc. As seguintes versões foram utilizadas no desenvolvimento do projeto:

- Versão Ubuntu: Ubuntu 14.04.6 LTS
- Versão gcc: gcc (Ubuntu 4.8.4-2ubuntu1 14.04.4) 4.8.4
- Versão flex: flex 2.5.35
- Versão bison: bison (GNU Bison) 3.0.2

Após conferir a instalação das dependências, navegue para o diretório raiz do programa. Nele, recomenda-se a execução do comando make para a instalação do projeto:

```
1   $ make
```

Mas caso não seja possível, analise o arquivo *Makefile* para entender como é realizado o procedimento de instalação e execute comando por comando.

Com o CWAM-Prolog instalado, você encontrará no diretório as pastas *bin*, *obj*, *prologs* e *src*. A pasta *prologs* contém todos os arquivos de teste selecionados. A pasta *obj* contém os objetos criados durante o processo de instalação e o diretório *bin* contém o executável. Os códigos fontes encontram-se no diretório *src*, nele você encontrara os subdiretórios *lexical*, *parser* e *wam*.

Para executar o CWAM-Prolog instalado, utilize o terminal no diretório raiz da aplicação, passando como parâmetro o nome do arquivo Prolog de entrada:

```
$ ./bin/main ./prologs/basic.pl
```

Com esse comando pode-se realizar a execução dos arquivos de teste selecionados. O código desenvolvido deve funcionar com a maioria dos programas Prolog puro. No entanto, separaram-se arquivos de teste específicos que apresentam casos importantes. Foram criados oito arquivos para realizar testes, dois corretos e dois com erros para cada tipo de erro: léxico, sintático e semântico. Os nomes dos arquivos são os seguintes:

- basic.pl
- list.pl
- basic_error.pl
- list_error.pl
- parser_error_1.pl
- parser_error_2.pl
- semantic_warning_1.pl
- semantic_warning_2.pl

6.1 Testes Corretos

O arquivo *basic.pl* contém um programa em Prolog básico utilizado para verificar se alunos estão matriculados em aula, professores que dão as matérias e quais professores dão aula para quais alunos. Além disso comentários e espaço em branco são inseridos para testar otimizações do analisador. Por outro lado, o arquivo *list.pl* já contém uma gramática mais complexa, utilizando operações com lista para verificar se a lista é um palíndromo. O resultado esperado para esses dois programas são saídas corretas com as árvores sintáticas com anotações semânticas e as tabelas de símbolos. Posteriormente, após a execução correta, o programa continuará executando para permitir que o usuário insira as devidas pesquisas para o programa.

Testando a Unificação

No entanto, os arquivos supracitados não funcionarão para testar a implementação no estado de desenvolvimento atual. Por isso, um arquivo de teste adicional *unification.pl* foi desenvolvido com o propósito de validar o processo de unificação. Sua execução deve mostrar a árvore sintática, a tabela de símbolos, e posteriormente irá aguardar por uma entrada de pesquisa.

A pesquisa que deve ser inserida é *p(Z, h(Z,W), f(W))*. Essa pesquisa vai ser unificada com o programa Prolog e a saída esperada é a *heap* de execução após o processo de unificação, representada na Figura 6.1. Se seguirmos o processo de desreferenciação na *heap* gerada, obteremos os seguintes resultados para as variáveis: $W = f(a)$, $X = f(a)$, $Y = f(f(a))$, $Z = f(f(a))$.

key	value
0	<STR, 1>
1	h/2
2	<REF, 13>
3	<REF, 16>
4	<STR, 5>
5	f/1
6	<REF, 3>
7	<STR, 8>
8	p/3
9	<REF, 2>
10	<STR, 1>
11	<STR, 5>
12	<STR, 13>
13	f/1
14	<REF, 3>
15	<STR, 16>
16	f/1:2
17	<REF, 19>
18	<STR, 19>
19	a/0

Figura 3. Representação da *heap* após processo de unificação.

6.2 Testes com Erros Léxicos

Em relação aos programas com erros léxicos, o programa *basic_error.pl* apresenta os seguintes erros, observe que o erro demonstra o nome do arquivo em que ocorreu, a linha e a coluna respectivamente:

```
ERROR: basic_error.pl:2:30:
Lexical error: Unrecognized Token Found: ;

ERROR: basic_error.pl:6:16:
Lexical error: Unrecognized Token Found: -

ERROR: basic_error.pl:18:7:
Lexical error: Unrecognized Token Found: {

ERROR: ../prologs/basic_error.pl:18:19:
Lexical error: Unrecognized Token Found: }
```

Esses erros representam, ponto e vírgula no lugar de ponto final, hífen no meio de uma regra e uso de chaves ao invés de parênteses respectivamente.

Por outro lado, arquivo *list_error.pl* deve apresentar os seguintes erros:

```
ERROR: ../prologs/list_error.pl:3:23:
Lexical error: Unrecognized Token Found: /

ERROR: ../prologs/list_error.pl:18:23:
Lexical error: Unrecognized Token Found: :=
```

Esses erros representam "/" no lugar do operador "|" e símbolo de regra escrito de forma incorreta ":=".

6.3 Testes com Erros Sintáticos

Adicionalmente, dois arquivos de teste considerando somente erros sintáticos foram escritos *parser_error_1.pl* e *parser_error_2.pl*.

Durante a execução dos arquivos de teste do analisador sintático, observe que o segundo arquivo de teste é o primeiro com os erros demonstrados pelo compilador, corrigidos. No entanto, o segundo arquivo ainda tem erros, que não aparecem no primeiro. Esse comportamento é um exemplo da limitação da política de erros escolhida. Portanto, os seguintes erros devem ser apresentados para os arquivos de teste *parser_error_1.pl* e *parser_error_2.pl* respectivamente:

```
ERROR: ./prologs/parser_error_1.pl:3:0:
syntax error, unexpected CON, expecting RULE_SYM or '.'

ERROR: ./prologs/parser_error_1.pl:6:0:
syntax error, unexpected VAR, expecting ']' or '|'
```

```
ERROR: ./prologs/parser_error_2.pl:3:0:
syntax error, unexpected VAR, expecting ')'

ERROR: ./prologs/parser_error_2.pl:6:0:
syntax error, unexpected ',', expecting ']'
```

6.4 Testes com Erros Semânticos

Com a árvore sintática anotada, e todas as funcionalidades relativas à semântica implementadas, pode-se testar a implementação de verificação estática no CWAM-Prolog. Todos os arquivos de erro apresentados anteriormente podem possivelmente apresentar erros semânticos, no entanto, para testar a implementação de forma eficiente, arquivos com erros específicos da semântica foram desenvolvidos.

Os arquivos de teste da análise semântica apresentam os dois erros semânticos implementados. O *semantic_warning_1.pl* se preocupa somente com casos de descontinuidade de predicados, enquanto o arquivo *semantic_warning_2.pl* cuida de todos os possíveis casos descritos de ocorrência de *singletons*. Os seguintes erros devem ser apresentados para ambos os arquivos, respectivamente:

```
WARNING: ./prologs/semantic_warning_1.pl:8:
semantic error, clauses of estuda/2
are not together in the source-file

WARNING: ./prologs/semantic_warning_1.pl:10:
semantic error, clauses of ensina/2
are not together in the source-file
```

```
WARNING: ./prologs/semantic_warning_2.pl:2:
semantic error, singleton variables: [X]

WARNING: ./prologs/semantic_warning_2.pl:4:
```

```
    semantic error, singleton variables: [X]

    WARNING: ./prologs/semantic_warning_2.pl:7:
    semantic error, singleton-marked variable
    appears more than once: _X
```

7 Considerações Finais

A implementação do CWAM-Prolog não demanda muito esforço na análise léxica, sintática e semântica. As maiores dificuldades encontradas foram relacionadas à manipulação de strings e estruturas de dados mais complexas, já que o C não providencia mecanismos de alto nível para manipulação.

Em relação aos erros semânticos, encontrou-se uma dificuldade especial em encontrá-los utilizando apenas a tabela de símbolos. Para contornar essa dificuldade precisou-se utilizar tabelas *hash* adicionais.

O mais complicado é em relação a implementação da Máquina Abstrata, apesar de Ait-Kaci, Hassan (1991) [2] descrever a implementação, muitos algoritmos são complicados e não são aprofundados. A linguagem C também apresenta diversas complicações ao implementar.

Referências

1. Aho, A.V., Sethi, R., Ullman, J.D., Lam, M.S.: Compilers: Principles, techniques, and tools. Pearson Education (2006)
2. Ait-Kaci, H.: Warren's abstract machine: a tutorial reconstruction. MIT press Cambridge (1991)
3. Lloyd, J.W.: Foundations of logic programming. Springer Science & Business Media (2012)
4. Matuszek, D.: A concise introduction to prolog (2012), `https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Prolog.html`, acessado: 17-03-2020
5. Paxon, V., Estes, W., Millaway, J.: Lexical analysis with flex, edition 2.5.35, the regents of the university of california, 2008
6. Sicstus prolog user's manual (1998), `https://sicstus.sics.se/documentation.html`, acessado: 17-03-2020
7. Warren, D.H.: An abstract prolog instruction set. Technical note 309 (1983)